

Join the discussion @ p2p.wrox.com



Wrox Programmer to Programmer™

.NET开发经典名著



Beginning C# 6 Programming with Visual Studio 2015

C#入门经典(第7版)

C# 6.0 & Visual Studio 2015

Benjamin Perkins
[美] Jacob Vibe Hammer 著
Jon D. Reid
齐立波 黄俊伟 译

清华大学出版社



C#入门经典 (第7版)

C# 6.0 & Visual Studio 2015

[美] Benjamin Perkins 著
Jacob Vibe Hammer
Jon D. Reid
齐立波 黄俊伟 译

清华大学出版社

北 京

作者简介

Benjamin Perkins（MCSD、MBA、ITIL）目前在微软（德国慕尼黑）工作，是IIS、ASP.NET和Azure应用服务高级技术顾问。他在IT行业工作了二十多年。他11岁时就开始在Atari 1200XL台式电脑上用QBASIC编写计算机程序。他喜爱诊断和排除技术问题，品味写出好程序的乐趣。上完高中后，他加入美国军队。在成功服完兵役后，他进入得克萨斯州的德克萨斯A&M大学，在那里获得管理信息系统的工商管理学士学位。

他在IT行的足迹遍及整个行业，包括程序员、系统架构师、技术支持工程师、团队领导和中层管理。在受雇于惠普时，他获得了众多奖项、学位和证书。他对技术和客户服务富有激情，期待排除故障，编写出更多世界级技术解决方案。

“我的方法是烂熟于心之后才编写代码，完整、正确地编写一次，这样就不需要再次考虑它，除非要改进它。”

Benjamin与妻子Andrea以及两个可爱的孩子Lea和Noa一起快乐地生活。

Jacob Vibe Hammer 是Kamstrup的一名软件架构师和开发人员，帮助公司为大型公用设施开发世界级智能网格解决方案。自他能拼写Basic之时，就开始了自己的编程生涯，Basic也是他使用的第一门编程语言。从那以后，他用过多种编程语言和解决方案架构。但进入21世纪

后，他主要在.NET平台上工作。如今，他主要编写C#和WPF程序，以及试用NoSQL数据库。Jacob是丹麦人，与妻儿一起居住在丹麦奥尔胡斯市。

Jon D. Reid 担任IFS Metrix Service Management (www.IFSWORLD.com) 的产品解决方案经理。他已与他人合著了多本.NET图书，包括*Beginning Visual C# 2010*、*Fast Track C#* 和*Pro Visual Studio .NET* 等。

C# 6和Visual Studio 2015编程实战指南

《C#入门经典》系列是屡获殊荣的C#名著和超级畅销书。最新版的《C#入门经典（第7版）C# 6.0 & Visual Studio 2015》全面介绍使用C# 6和.NET Framework编写程序的基础知识，是编程新手的理想读物。这本分步讲解的使用教程从最基本的面向对象编程讲起，浓墨重彩地描述初学者最常用的工具，不要求读者居有任何编程经验。紧贴使用的示例使用Visual Studio 2015中的C#环境，涵盖微软为使C#更好兼容其他编程语言所做的最新改进。本书呈现微软资深开发人员的专家级建议，将指导初学者立即上手编写Windows和Web应用程序。

主要内容

首先讲解编程基础知识，如变量、流控制、面向对象编程、类、函数、集合、比较和转换等。

重点介绍Visual Studio 2015中初学者喜欢的C# 6开发环境，囊括所有最新功能和语言改进。

包括云和Windows编程中级内容，涵盖数据库和SML

揭密错误处理技术和调试过程

以专家撰写的分步指南为特色，指导初学者在真实编程环境中编写

有用的代码

源代码下载



Benjamin Perkins, Jacob Vibe Hammer, Jon D. Reid

Beginning C# 6 Programming with Visual Studio 2015

EISBN: 978-1-119-09668-9

Copyright © 2016 by John Wiley & Sons, Inc., Indianapolis, Indiana

All Rights Reserved. This translation published under license.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Visual C# is a registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

本书中文简体字版由Wiley Publishing, Inc.授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2016-1652

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书封面贴有**Wiley**公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：**010-62782989 13701121933**

图书在版编目（**CIP**）数据

C#入门经典（第7版）C# 6.0 & Visual Studio 2015 / （美）本杰明·帕金斯（Benjamin Perkins）等著；齐立波，黄俊伟 译．—北京：清华大学出版社，2016

（.NET开发经典名著）

书名原文：Beginning C# 6 Programming with Visual Studio 2015

ISBN 978-7-302-44406-0

I. ①C... II. ①本...②齐...③黄... III. ①C语言—程序设计
IV. ①TP312

中国版本图书馆CIP数据核字（2016）第168659号

责任编辑：王 军 韩宏志

装帧设计：牛静敏

责任校对：成凤进

责任印制:

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦A座

邮 编: 100084

社 总 机: 010-62770175

邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者:

装 订 者:

经 销: 全国新华书店

开 本: 185mm×260mm

印 张: 45

字 数: 1207千字

版 次: 2016年8月第1版

印 次： 2016年8月第1次印刷

印 数： 1~5000

定 价： 98.00元

产品编号：

译者序

C#是微软公司发布的一种面向对象的、运行在.NET Framework上的高级程序设计语言。C#几乎集中了所有关于软件开发和软件工程的最新成果：面向对象、类型安全、组件技术、自动内存管理、跨平台异常处理、版本控制、代码安全管理.....，是一种安全、稳定、简单、优雅、由C和C++衍生而来的面向对象的编程语言，综合了VB简单的可视化操作和C++的高运行效率优点，以其强大的操作能力、优雅的语法风格、创新的语言特性和便捷的对面向组件编程的支持成为.NET开发的首选语言。

Visual Studio（简称VS）是美国微软公司的开发工具包系列产品，是目前最流行的Windows平台应用程序的集成开发环境，它包括了整个软件生命周期中所需的大部分工具，如UML工具、代码管控工具、集成开发环境（IDE）等。Visual Studio最新版本为Visual Studio 2015，基于.NET Framework 4.6。

本书旨在介绍C#开发的基础知识。本书第 I 部分介绍C#语言的语法和用法，然后讨论较复杂的面向对象编程主题，第II部分将讲述Windows基本桌面编程和高级桌面编程。第III部分研究基于云的Web应用程序编程，第IV部分将讲述数据访问（对数据库、文件系统和XML数据的访问）和LINQ，第V部分将讨论WCF和通用应用。

本书采用循序渐进的编排方式，所以读者应能从头开始一直阅读到

最后。本书介绍如何使用C#编程，读者应自己输入所有的示例代码，再编译和执行输入的代码，而不是从下载文件中复制它们。这似乎很麻烦，但输入C#语句可以帮助理解C#，特别是觉得某些地方很难掌握时，自己输入代码就非常有帮助。如果例子无法运行，不要直接从书中查找原因，而应在自己输入的示例代码中找原因，这是编写C#代码时必须做的一项工作。

犯错也是学习过程中不可避免的，练习应提供大量犯错的机会，最好自己编几个练习题。如果不确定如何编写代码，应翻阅前面的内容。犯的错误越多，对C#的功能和错误的原因的认识就越深刻。读者应完成所有练习，除非肯定自己无法解决问题，否则不要看答案。许多练习都涉及某章内容的一个直接应用，换言之，它们仅是一种实践，但也有一些练习需要多动脑子，甚至需要一点灵感。

本书的读者不需要具备任何编程经验。但本书同样适合具有编程经验且希望进行Web程序设计的读者阅读。这些读者可能比较了解计算机知识，但未必掌握Web技术。另外，一些读者具备设计背景，但对计算机知识和Web技术不大了解。那么，本书可以作为一条进入编程和Web应用程序开发世界的快捷通道。对于所有读者，本书都物有所值。

在这里要感谢清华大学出版社的编辑，他们为本书的翻译投入了巨大的热情并付出了很多心血。没有他们的帮助和鼓励，本书不可能顺利付梓。

在翻译这本经典之作的过程中，译者在忠于原文的基础上力求做到“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。本书全部章节由齐立波、黄俊伟翻译，参与翻译的还有孔祥亮、陈跃华、杜思明、熊晓磊、曹汉鸣、陶晓云、

王通、方峻、李小凤、曹晓松、蒋晓冬、邱培强、洪妍、李亮辉、高娟妮、曹小震、陈笑。

最后，希望读者通过阅读本书能早日步入C#语言编程的殿堂，领略C#语言之美！

技术编辑简介

John Mueller是一位自由撰稿人和技术编辑。他用自己的心血编写99本书和600余篇文章。主题范围从联网到人工智能，从数据库管理到编程入门知识，非常广泛。一些正在发行的图书包括用于初学者的Python、用于数据科学家的Python和MATLAB等主题。他还编写了一个Java电子学习套件、一本关于用JavaScript开发HTML5的书籍，和一本关于CSS3的图书。作为技术编辑，他曾帮助60多名作者修订手稿。John还为*Data Based Advisor* 和*Coast Compute* 杂志提供技术编辑服务。John的博客网址是[http://blog.johnmueller books.com/](http://blog.johnmuellerbooks.com/)。

致谢

为使本书内容以清晰美观的形式呈现给学生和专业人士，使他们从中获益，需要做大量的工作。作者的确有卓越的技术知识和经验供大家分享，但如果没有技术作家、技术评审人员、开发人员、编辑、出版人员、平面设计师等提供有价值的帮助，就不可能编写出高质量的书籍。编程技术日新月异，在有效技术过时之前，个人无法凭一己之力完成所有这些任务。正因为如此，作者只有与伟大的团队合作，才能很快把本书的所有组件组合在一起，才能确保把最新信息传达给读者，帮助读者了解最新功能。感谢Kelly Talbot很好地完成了项目管理和全书的技术评审工作，感谢John Mueller在整个过程的技术审查和建议。最后，感谢在幕后帮助本书出版的所有人员。

前言

C#是Microsoft于2000年7月推出.NET Framework的第1版时提供的一种全新语言。C#从那时起迅速流行开来，成为使用.NET Framework的桌面、Web和云开发人员无可争议的选择。他们喜欢C#的一个原因是其继承自C/C++的简洁明了的语法，这种语法简化了以前给程序员带来困扰的一些问题。尽管做了这些简化，但C#仍保持了C++原有的功能，所以现在没理由不从C++转向C#。C#语言并不难，也非常适合学习基本编程技术。易于学习，再加上.NET Framework的功能，使C#成为开始你编程生涯的绝佳方式。

C#的最新版本C# 6是.NET Framework 4.6的一部分，它建立在已有的成功基础之上，还添加了一些更吸引人的功能。Visual Studio的最新版本Visual Studio 2015和开发工具的Visual Studio Express/Community 2015系列也有许多变化和改进，这大大简化了编程工作，显著提高了效率。

本书将全面介绍C#编程的所有知识，从该语言本身一直到桌面编程和云编程，再到数据源的使用，最后是一些新的高级技术。我们还将学习Visual Studio 2015的功能和利用它开发应用程序的各种方式。

本书文笔优美流畅，阐述清晰，每一章都以前面章节的内容为基础，便于读者掌握高级技术。每个概念都会根据需要来介绍和讨论，而不会突然冒出某个技术术语来妨碍读者的阅读和理解。本书尽量减少使

用的技术术语数量，但如有必要，将根据上下文进行正确的定义和布置。

本书作者都是各自领域的专家，都是C#语言和.NET Framework的爱好者，没人比他们更有资格讲授C#了，他们将在你掌握从基本规则到高级技术的过程中为你保驾护航。除了基础知识外，本书还有许多有益的提示、练习、完全成熟的示例代码（可从p2p.wrox.com下载），在你的职业生涯中一定会反复用到它们。

本书将毫无保留地传授这些知识，希望读者能通过阅读本书成为最优秀的程序员。

0.1 本书读者对象

本书面向想学习如何使用.NET Framework编写C#程序的所有人。本书针对的是想要通过学习一种干净、现代、优雅的编程语言来掌握程序设计的完完全全的初学者。但是，对于熟悉其他语言、想要探索.NET平台的人们，以及想要了解.NET使用的旗舰语言的.NET开发人员，本书同样有用。

0.2 本书内容

本书前面的章节介绍C#语言本身，读者不需要具备任何编程经验。以前对其他语言有一定了解的开发人员，会觉得这些章节的内容非常熟悉。C#语法的许多方面都与其他语言相同，许多结构对所有的编程语言来说都是相通的（例如，循环和分支结构）。但是，即使是有经验的程序员也可以通过这些章节理解此类技术应用于C#的特征，从而从中获益。

如果读者是编程新手，就应从头开始学习，了解基本的编程概念，并熟悉C#和支持C#的.NET Framework平台。如果读者对.NET Framework比较陌生，但知道如何编程，就应阅读第1章，然后快速跳读后面几章，这样就能掌握C#语言的应用方式了。如果读者知道如何编程，但以前从未接触过面向对象的编程语言，就应从第8章开始阅读以后的章节。

如果读者对C#语言比较了解，就可以集中精力学习那些详细论述最新.NET Framework和C#语言开发的章节，尤其是集合、泛型和C#语言的新增内容（第11章～第13章），或者完全跳过本书第I部分，从第14章开始学习。

本书章节的编排方式可以达到两个目的：可以按顺序阅读这些章节，将其视为C#语言的一个完整教程；还可以按照需要深入学习这些章节，将其作为一本参考资料。

除核心内容外，从第3章开始，每章末尾还包含一组习题，完成这些习题有助于读者理解所学的内容。习题包括简单的选择题、判断题以

及需要修改或建立应用程序的较难问题。附录A给出了全部习题的答案。

本书特别注重与C# 6、.NET 4.6的一致性。每一章都进行了彻底的检查，删掉了不太相关的内容，增加了新材料。所有代码都在最新版本的开发工具上进行了测试，所有屏幕截图都在Windows 8.1/10上重新截取，以提供最新的窗口和对话框。

本书的亮点包括：

- 增加并改进了代码示例。
- 涵盖C# 6和.NET 4.6的所有新内容，包括如何创建通用Windows应用程序。
- 增加了编写云应用程序的示例，并使用Azure SDK创建和访问云资源。

0.3 本书结构

本书分为6个部分。

- 前言：概述本书的内容。
- **OOP语言**：介绍C#语言的所有内容，从基础知识到面向对象的技术，一应俱全。
- **Windows编程**：介绍如何用WPF库编写和部署桌面应用程序。
- 云编程：描述云应用程序的开发和部署，包括Web API的创建和使用。
- 数据访问：介绍如何在应用程序中使用数据，包括存储在硬盘文件中的数据、以XML格式存储的数据和数据库中的数据。
- 其他技术：讲述使用C#和.NET Framework的一些额外方式，包括WCF和通用Windows应用程序。

下面介绍本书5个重要部分中的章节。

0.3.1 OOP语言（第1章～第13章）

第1章介绍C#及其与.NET的关系，了解在这个环境下编程的基础知识，以及Visual Studio 2015（VS）与它的关系。

第2章开始介绍如何编写C#应用程序，学习C#的语法，并将C#和示例命令行、Windows应用程序结合起来使用。这些示例将说明C#如何快速轻松地启动和运行，并附带介绍VS开发环境以及本书将要使用的基

本窗口和工具。

接着将学习C#的基础知识。第3章介绍变量的含义以及如何操纵它们。第4章将用流程控制（循环和分支）改进应用程序的结构，第5章介绍一些高级变量类型，如数组。第6章开始以函数形式封装代码，这样就更易于执行重复操作，使代码更容易让人理解。

从第7章开始将运用C#语言的基础知识，调试应用程序。这包括在运行应用程序时输出跟踪信息，使用VS查找错误，在强大的调试环境中找出解决问题的办法。

第8章将学习面向对象编程（Object-Oriented Programming, OOP）。首先了解这个术语的含义，回答“什么是对象”？OOP初看起来是较难的问题。我们将用一整章的篇幅来介绍它，解释对象的强大之处。直到该章的最后才会真正使用C#代码。

第9章将理论知识应用于实践，开始在C#应用程序中使用OOP时，这才体现出C#的真正威力。在第9章介绍如何定义类和接口之后，第10章将探讨类成员（包括字段、属性和方法），在这一章的最后将开始创建一个扑克牌游戏，这个游戏将在几章中开发完成，它非常有助于理解OOP。

学习了OOP在C#中的工作原理后，第11章将介绍几种常见的OOP场景，包括处理对象集合、比较和转换对象。第12章讨论.NET 2.0中引入的一个非常有用的C#特性——泛型，利用它可以创建非常灵活的类。第13章通过一些其他技术（主要是事件，它在Windows编程中非常重要）继续讨论C#语言和OOP。最后介绍C#在3.0、4、5和6版本中引入的新特性。

0.3.2 Windows编程（第14章和第15章）

第14章开始介绍Windows编程概念，理解在VS中如何实现Windows编程。该章主要关注如何使用WPF以图形化方式构建桌面应用程序，以及用最少的时间和精力创建高级桌面应用程序。你将首先学习WPF编程的基础知识，然后在该章和第15章逐渐拓展相关知识。第15章介绍在应用程序中如何使用.NET Framework提供的丰富控件。

0.3.3 云编程（第16章和第17章）

第16章首先描述云编程，再讨论云优化堆栈。云环境不同于传统的程序编码方式，所以讨论、定义了几个云编程模式。为完成这一章，需要一个免费的Azure账户，以便创建一个App Services Web App，然后使用Azure SDK和C#，在ASP.NET 4.6 Web应用程序中创建和访问存储账户。

第17章将学习如何创建ASP.NET Web API，并部署到云中，然后在类似的ASP.NET 4.6 Web应用程序中使用Web API。这一章最后讨论云中两个最有价值的特性：硬件资源的缩放和最优利用方式。

0.3.4 数据访问（第18章～第21章）

第18章介绍应用程序如何将数据保存到磁盘以及如何检索磁盘上的数据（作为简单的文本文件或者更复杂的数据表示方式）。该章还将讨

论如何压缩数据，如何监视和处理文件系统的变化。

第19章学习数据交换的事实标准XML，简要论述JSON格式。之前的章节接触过XML几次，而该章将讨论XML的基本规则，论述XML的所有功能。

该部分其余章节介绍LINQ（这是内置于.NET Framework最新版本中的查询语言）。第20章简要介绍LINQ。第21章讨论如何使用LINQ访问数据库和其他数据。

0.3.5 其他技术（第22章和第23章）

第22章简要介绍Windows Communication Foundation（WCF），它为在企业级以编程方式跨本地网络和Internet访问信息和功能提供了许多工具。该章将介绍如何以平台无关的方式使用WCF，向Web应用程序和桌面应用程序公开复杂的数据和功能。

第23章展示如何创建通用Windows应用程序，这是Windows新增的。本章建立在第14和第15章的基础上，介绍如何创建可以运行在所有Windows平台上的Windows应用程序。

0.4 使用本书的要求

本书中C#和.NET Framework的代码和描述都适用于C# 6和.NET 4.6。除了Framework之外，不需要其他组件就可以理解本书的这个方面，但许多示例都需要使用开发工具。本书将Visual Studio 2015作为主要开发工具，但是，如果没有安装此工具，可以使用免费的Visual Studio Express/Community 2015产品系列。在本书的第I部分，可使用Visual Studio Express/Community 2012 for Windows Desktop来创建桌面和控制台应用程序。对于其余章节，可使用Visual Studio Express/Community 2015 for Windows 10创建通用Windows应用程序，使用Visual Studio Express/Community 2015 for Cloud创建云应用程序，并在需要访问数据库的应用程序中使用SQL Server Express 2014。一些功能只能在Visual Studio 2015中使用，但这不会妨碍练习本书的示例。

0.5 本书约定

为了帮助读者在阅读本书的过程中获取最多信息，并随时了解当前处理的事项，本书使用了许多约定。

警告： 带有警告图标的方框包含了重要且应该记住的信息，这些信息与周围的文字直接相关联。

提示： 带有铅笔图标的方框表示注释、提示、暗示、技巧或对当前讨论的弦外之音。

本书通过两种方式来显示代码：

- 对于大多数代码示例，使用没有突出显示的等宽字体来表示。
- 对在当前上下文中特别重要的代码，用粗体字来强调显示。

0.6 勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果你在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

请给wkservice@vip.163.com发电子邮件，我们就会检查你的反馈信息，如果是正确的，我们将在本书的后续版本中采用。

要在网站上找到本书英文版的勘误表，可以登录<http://www.wrox.com>，通过Search工具或书名列表查找本书，然后在本书的细目页面上，单击Book Errata链接。在这个页面上可以查看到Wrox编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是www.wrox.com/misc-pages/booklist.shtml。

0.7 p2p.wrox.com

要与作者和同行讨论，请加入p2p.wrox.com上的P2P论坛。这个论坛是一个基于Web的系统，便于你张贴与Wrox图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有了新的消息时，它可以给你传送感兴趣的论题。Wrox作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在<http://p2p.wrox.com>上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

- (1) 进入p2p.wrox.com，单击Register链接。
- (2) 阅读使用协议，并单击Agree按钮。
- (3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，单击Submit按钮。
- (4) 你会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

提示：

不加入P2P也可以阅读论坛上的消息，但要张贴自己的消息，就必须加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在Web上阅读消息。如果要想让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的Subscribe to this Forum图标。

关于使用Wrox P2P的更多信息，可阅读P2P FAQ，了解论坛软件的工作情况以及P2P和Wrox图书的许多常见问题。要阅读FAQ，可以在任意P2P页面上单击FAQ链接。

0.8 源代码

在读者学习本书中的示例时，可以手工输入所有的代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点<http://www.wrox.com/>下载。登录站点<http://www.wrox.com/>，使用Search工具或使用书名列表就可以找到本书。接着单击本书细目页面上的Download Code链接，就可以获得所有的源代码。也可以访问www.tupwk.com.cn/downpage，输入本书中文书名或中文ISBN，下载各章的源代码。

提示：

由于许多图书的标题都很类似，所以按ISBN搜索是最简单的，本书英文版的ISBN是978-1-119-09668-9。

下载代码后，只需用自己喜欢的解压缩软件对它进行解压缩即可。另外，也可以进入<http://www.wrox.com/dynamic/books/download.aspx>上的Wrox代码下载主页，查看本书和其他Wrox图书的所有代码。

目 录

[前言](#)

[0.1 本书读者对象](#)

[0.2 本书内容](#)

[0.3 本书结构](#)

[0.3.1 OOP语言（第1章～第13章）](#)

[0.3.2 Windows编程（第14章和第15章）](#)

[0.3.3 云编程（第16章和第17章）](#)

[0.3.4 数据访问（第18章～第21章）](#)

[0.3.5 其他技术（第22章和第23章）](#)

[0.4 使用本书的要求](#)

[0.5 本书约定](#)

[0.6 勘误表](#)

[0.7 p2p.wrox.com](#)

[0.8 源代码](#)

[第 I 部分 OOP语言](#)

[第1章 C#简介](#)

[1.1 .NET Framework的含义](#)

[1.1.1 .NET Framework的内容](#)

[1.1.2 使用.NET Framework编写应用程序](#)

[1.2 C#的含义](#)

[1.2.1 用C#能编写什么样的应用程序](#)

[1.2.2 本书中的C#](#)

[1.3 Visual Studio 2015](#)

[1.3.1 Visual Studio Express 2015产品](#)

[1.3.2 解决方案](#)

[1.4 本章要点](#)

[第2章 编写C#程序](#)

[2.1 Visual Studio 2015开发环境](#)

[2.2 控制台应用程序](#)

[2.2.1 Solution Explorer窗口](#)

[2.2.2 Properties窗口](#)

[2.2.3 Error List窗口](#)

[2.3 桌面应用程序](#)

[2.4 本章要点](#)

[第3章 变量和表达式](#)

[3.1 C#的基本语法](#)

[3.2 C#控制台应用程序的基本结构](#)

[3.3 变量](#)

[3.3.1 简单类型](#)

[3.3.2 变量的命名](#)

[3.3.3 字面值](#)

[3.4 表达式](#)

[3.4.1 数学运算符](#)

[3.4.2 赋值运算符](#)

[3.4.3 运算符的优先级](#)

[3.4.4 名称空间](#)

[3.5 练习](#)

[3.6 本章要点](#)

[第4章 流程控制](#)

[4.1 布尔逻辑](#)

[4.1.1 布尔按位运算符和赋值运算符](#)

[4.1.2 运算符优先级的更新](#)

[4.2 分支](#)

[4.2.1 三元运算符](#)

[4.2.2 if语句](#)

[4.2.3 switch语句](#)

[4.3 循环](#)

[4.3.1 do循环](#)

[4.3.2 while循环](#)

[4.3.3 for循环](#)

[4.3.4 循环的中断](#)

[4.3.5 无限循环](#)

[4.4 练习](#)

[4.5 本章要点](#)

[第5章 变量的更多内容](#)

[5.1 类型转换](#)

[5.1.1 隐式转换](#)

[5.1.2 显式转换](#)

[5.1.3 使用Convert命令进行显式转换](#)

[5.2 复杂的变量类型](#)

[5.2.1 枚举](#)

[5.2.2 结构](#)

[5.2.3 数组](#)

[5.3 字符串的处理](#)

[5.4 练习](#)

[5.5 本章要点](#)

[第6章 函数](#)

[6.1 定义和使用函数](#)

[6.1.1 返回值](#)

[6.1.2 参数](#)

[6.2 变量的作用域](#)

[6.2.1 其他结构中变量的作用域](#)

[6.2.2 参数和返回值与全局数据](#)

[6.3 Main\(\)函数](#)

[6.4 结构函数](#)

[6.5 函数的重载](#)

[6.6 委托](#)

[6.7 练习](#)

[6.8 本章要点](#)

[第7章 调试和错误处理](#)

[7.1 Visual Studio中的调试](#)

[7.1.1 非中断（正常）模式下的调试](#)

[7.1.2 中断模式下的调试](#)

[7.2 错误处理](#)

[7.2.1 try...catch...finally](#)

[7.2.2 列出和配置异常](#)

[7.3 练习](#)

[7.4 本章要点](#)

[第8章 面向对象编程简介](#)

[8.1 面向对象编程的含义](#)

[8.1.1 对象的含义](#)

[8.1.2 一切皆对象](#)

[8.1.3 对象的生命周期](#)

[8.1.4 静态成员和实例类成员](#)

[8.2 OOP技术](#)

- [8.2.1 接口](#)
- [8.2.2 继承](#)
- [8.2.3 多态性](#)
- [8.2.4 对象之间的关系](#)
- [8.2.5 运算符重载](#)
- [8.2.6 事件](#)
- [8.2.7 引用类型和值类型](#)

[8.3 桌面应用程序中的OOP](#)

[8.4 练习](#)

[8.5 本章要点](#)

[第9章 定义类](#)

[9.1 C#中的类定义](#)

[9.2 System.Object](#)

[9.3 构造函数和析构函数](#)

[9.4 Visual Studio中的OOP工具](#)

[9.4.1 Class View窗口](#)

[9.4.2 对象浏览器](#)

[9.4.3 添加类](#)

[9.4.4 类图](#)

[9.5 类库项目](#)

[9.6 接口和抽象类](#)

[9.7 结构类型](#)

[9.8 浅度和深度复制](#)

[9.9 练习](#)

[9.10 本章要点](#)

[第10章 定义类成员](#)

[10.1 成员定义](#)

[10.1.1 定义字段](#)

[10.1.2 定义方法](#)

[10.1.3 定义属性](#)

[10.1.4 重构成员](#)

[10.1.5 自动属性](#)

[10.2 类成员的其他主题](#)

[10.2.1 隐藏基类方法](#)

[10.2.2 调用重写或隐藏的基类方法](#)

[10.2.3 嵌套的类型定义](#)

[10.3 接口的实现](#)

[10.4 部分类定义](#)

[10.5 部分方法定义](#)

[10.6 示例应用程序](#)

[10.6.1 规划应用程序](#)

[10.6.2 编写类库](#)

[10.6.3 类库的客户应用程序](#)

[10.7 Call Hierarchy窗口](#)

[10.8 练习](#)

[10.9 本章要点](#)

[第11章 集合、比较和转换](#)

[11.1 集合](#)

[11.1.1 使用集合](#)

[11.1.2 定义集合](#)

[11.1.3 索引符](#)

[11.1.4 给CardLib添加Cards集合](#)

[11.1.5 键控集合和IDictionary](#)

[11.1.6 迭代器](#)

[11.1.7 迭代器和集合](#)

[11.1.8 深度复制](#)

[11.1.9 给CardLib添加深度复制](#)

[11.2 比较](#)

[11.2.1 类型比较](#)

[11.2.2 值比较](#)

[11.3 转换](#)

[11.3.1 重载转换运算符](#)

[11.3.2 as运算符](#)

[11.4 练习](#)

[11.5 本章要点](#)

[第12章 泛型](#)

[12.1 泛型的含义](#)

[12.2 使用泛型](#)

[12.2.1 可空类型](#)

[12.2.2 System.Collections.Generic名称空间](#)

[12.3 定义泛型类型](#)

[12.3.1 定义泛型类](#)

[12.3.2 定义泛型接口](#)

[12.3.3 定义泛型方法](#)

[12.3.4 定义泛型委托](#)

[12.4 变体](#)

[12.4.1 协变](#)

[12.4.2 抗变](#)

[12.5 练习](#)

[12.6 本章要点](#)

[第13章 高级C#技术](#)

[13.1 ::运算符和全局名称空间限定符](#)

[13.2 定制异常](#)

[13.3 事件](#)

[13.3.1 事件的含义](#)

[13.3.2 处理事件](#)

[13.3.3 定义事件](#)

[13.4 扩展和使用CardLib](#)

[13.5 特性](#)

[13.5.1 读取特性](#)

[13.5.2 创建特性](#)

[13.6 初始化器](#)

[13.6.1 对象初始化器](#)

[13.6.2 集合初始化器](#)

[13.7 类型推理](#)

[13.8 匿名类型](#)

[13.9 动态查找](#)

[13.10 高级方法参数](#)

[13.10.1 可选参数](#)

[13.10.2 命名参数](#)

[13.11 Lambda表达式](#)

[13.11.1 复习匿名方法](#)

[13.11.2 把Lambda表达式用于匿名方法](#)

[13.11.3 Lambda表达式的参数](#)

[13.11.4 Lambda表达式的语句体](#)

[13.11.5 Lambda表达式用作委托和表达式树](#)

[13.11.6 Lambda表达式和集合](#)

[13.12 练习](#)

[13.13 本章要点](#)

[第II部分 Windows编程](#)

[第14章 基本桌面编程](#)

[14.1 XAML](#)

[14.1.1 关注点分离](#)

[14.1.2 XAML基础知识](#)

[14.2 动手实践](#)

[14.2.1 WPF控件](#)

[14.2.2 属性](#)

[14.2.3 事件](#)

[14.3 控件布局](#)

[14.3.1 堆叠顺序](#)

[14.3.2 对齐、边距、填充和尺寸](#)

[14.3.3 Border控件](#)

[14.3.4 Canvas控件](#)

[14.3.5 DockPanel控件](#)

[14.3.6 StackPanel控件](#)

[14.3.7 WrapPanel控件](#)

[14.3.8 Grid控件](#)

[14.4 游戏客户端](#)

[14.4.1 About窗口](#)

[14.4.2 Options窗口](#)

[14.4.3 数据绑定](#)

[14.4.4 使用ListBox控件启动游戏](#)

[14.5 练习](#)

[14.6 本章要点](#)

[第15章 高级桌面编程](#)

[15.1 主窗口](#)

[15.1.1 菜单控件](#)

[15.1.2 路由命令和菜单](#)

[15.2 创建控件并设置样式](#)

[15.2.1 样式](#)

[15.2.2 模板](#)

[15.2.3 值转换器](#)

[15.2.4 触发器](#)

[15.2.5 动画](#)

[15.3 WPF用户控件](#)

[15.4 把所有内容结合起来](#)

[15.4.1 重构域模型](#)

[15.4.2 视图模型](#)

[15.4.3 大功告成](#)

[15.5 练习](#)

[15.6 本章要点](#)

[第III部分 云编程](#)

[第16章 基本的云编程](#)

[16.1 云、云编程和云优化堆栈](#)

[16.2 云模式和最佳实践](#)

[16.3 使用Microsoft Azure C#库创建存储容器](#)

[16.4 创建使用存储容器的ASP.NET 4.6网站](#)

[16.5 练习](#)

[16.6 本章要点](#)

[第17章 高级云编程和部署](#)

[17.1 创建ASP.NET Web API](#)

[17.2 在Microsoft Azure上部署和使用ASP.NET Web API](#)

[17.3 扩展Microsoft Azure平台上的ASP.NET Web API](#)

[17.4 练习](#)

[17.5 本章要点](#)

[第IV部分 数据访问](#)

[第18章 文件](#)

[18.1 用于输入和输出的类](#)

[18.1.1 File类和Directory类](#)

[18.1.2 FileInfo类](#)

[18.1.3 DirectoryInfo类](#)

[18.1.4 路径名和相对路径](#)

[18.2 流](#)

[18.2.1 使用流的类](#)

[18.2.2 FileStream对象](#)

[18.2.3 StreamWriter对象](#)

[18.2.4 StreamReader对象](#)

[18.2.5 异步文件访问](#)

[18.2.6 读写压缩文件](#)

[18.3 监控文件系统](#)

[18.4 练习](#)

[18.5 本章要点](#)

[第19章 XML和JSON](#)

[19.1 XML基础](#)

[19.2 JSON基础](#)

[19.3 XML模式](#)

[19.4 XML文档对象模型](#)

[19.4.1 XmlDocument类](#)

[19.4.2 XmlElement类](#)

[19.4.3 修改节点的值](#)

[19.5 把XML转换为JSON](#)

[19.6 用XPath搜索XML](#)

[19.7 练习](#)

[19.8 本章要点](#)

[第20章 LINQ](#)

[20.1 使用LINQ to XML](#)

[20.1.1 LINQ to XML函数构造方式](#)

[20.1.2 处理XML片段](#)

[20.2 LINQ提供程序](#)

[20.3 LINQ查询语法](#)

[20.3.1 用var关键字声明结果变量](#)

[20.3.2 指定数据源：from子句](#)

[20.3.3 指定条件：where子句](#)

[20.3.4 选择元素：select子句](#)

[20.3.5 完成：使用foreach循环](#)

[20.3.6 延迟执行的查询](#)

[20.4 LINQ方法语法](#)

[20.4.1 LINQ扩展方法](#)

[20.4.2 查询语法和方法语法](#)

[20.4.3 Lambda表达式](#)

[20.5 排序查询结果](#)

[20.6 orderby子句](#)

[20.7 查询大型数据集](#)

[20.8 使用聚合运算符](#)

[20.9 单值选择查询](#)

[20.10 多级排序](#)

[20.11 组合查询](#)

[20.12 Join查询](#)

[20.13 练习](#)

[20.14 本章要点](#)

[第21章 数据库](#)

[21.1 使用数据库](#)

[21.2 安装SQL Server Express](#)

[21.3 Entity Framework](#)

[21.4 Code First数据库](#)

[21.5 数据库的位置](#)

[21.6 导航数据库关系](#)

[21.7 处理迁移](#)

[21.8 在已有的数据库中创建和查询XML](#)

[21.9 练习](#)

[21.10 本章要点](#)

[第V部分 其他技术](#)

[第22章 Windows Communication Foundation](#)

[22.1 WCF的含义](#)

[22.2 WCF概念](#)

[22.2.1 WCF通信协议](#)

[22.2.2 地址、端点和绑定](#)

[22.2.3 协定](#)

[22.2.4 消息模式](#)

[22.2.5 行为](#)

[22.2.6 驻留](#)

[22.3 WCF编程](#)

[22.3.1 WCF测试客户端程序](#)

[22.3.2 定义WCF服务协定](#)

[22.3.3 自驻留的WCF服务](#)

[22.4 练习](#)

[22.5 本章要点](#)

[第23章 通用应用程序](#)

[23.1 入门](#)

[23.2 通用应用程序](#)

[23.3 应用程序概念和设计](#)

[23.3.1 屏幕方向](#)

[23.3.2 菜单和工具栏](#)

[23.3.3 磁贴和徽章](#)

[23.3.4 应用程序的生存期](#)

[23.3.5 锁屏应用程序](#)

[23.4 应用程序的开发](#)

[23.4.1 自适应显示](#)

[23.4.2 沙箱应用程序](#)

[23.4.3 在页面之间导航](#)

[23.4.4 CommandBar控件](#)

[23.4.5 管理状态](#)

[23.5 Windows Store应用程序的常见元素](#)

[23.6 Windows Store](#)

[23.6.1 打包应用程序](#)

[23.6.2 创建包](#)

[23.7 练习](#)

[23.8 本章要点](#)

[附录A 习题答案](#)

第 I 部分 OOP语言

- 第1章 C#简介
- 第2章 编写C#程序
- 第3章 变量和表达式
- 第4章 流程控制
- 第5章 变量的更多内容
- 第6章 函数
- 第7章 调试和错误处理
- 第8章 面向对象编程简介
- 第9章 定义类
- 第10章 定义类成员
- 第11章 集合、比较和转换
- 第12章 泛型

第13章 高级C#技术

第1章 C#简介

本章内容：

- .NET Framework
- .NET应用程序的工作原理
- C#的概念及其与.NET Framework的关系
- 用C#创建.NET应用程序的工具

本章源代码下载

本章源代码的下载网址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 1 Code后，可以找到与本章示例对应的单独文件。

本书第I部分将介绍使用最新版本的C# 语言所需的基础知识。第1章将概述C#和.NET Framework，包括这两项技术的含义、作用及相互关系。

首先讨论.NET Framework。这种技术包含的许多概念初看起来都不是很容易掌握。也就是说，我们必须在很短的篇幅里介绍许多新概念，但快速浏览这些基础知识对于理解如何利用C#进行编程是非常重要的，本书后面将详细论述这里提到的许多话题。

之后，本章将讨论C#本身，包括它的起源以及与C++的类似之处。最后介绍本书使用的主要工具：Visual Studio 2015（VS 2015）。VS 2015是Microsoft提供的一系列开发环境中最新的一个，本书将用到它的各种新功能（包括对Windows Store应用程序的完整支持）。

1.1 .NET Framework的含义

.NET Framework（现在最新版本是4.6）是Microsoft为开发应用程序而创建的一个具有革命意义的平台。这句话最有趣的地方在于它的广义性，但这是有原因的。首先，注意这句话没有说“在Windows操作系统上开发应用程序”。尽管.NET Framework的Microsoft版本运行在Windows操作系统和Windows Phone操作系统上，但它也有运行在其他操作系统上的版本，例如Mono，它是.NET Framework的开源版本（包含C#编译器），该版本可以运行在几个操作系统上，包括各种Linux版本和Mac OS，详见<http://www.mono-project.com>。另外，Mono还有一些版本可以运行在iPhone（MonoTouch）和Android（Mono for Android，也称为MonoDroid）智能手机上。使用.NET Framework的一个重要原因是它可以作为集成各种操作系统的方式。

另外，上面给出的.NET Framework定义并未限制应用程序的类型。这是因为本来就没有限制。可以使用.NET Framework创建桌面应用程序、Windows Store应用程序、云/Web应用程序、Web API和其他各种类型的应用程序。另外注意，对于Web、云和Web API应用程序，按照定义，它们是多平台的应用程序，因为任何带有Web浏览器的系统都可以访问它们。

.NET Framework的设计方式确保它可以用于各种语言，包括本书介绍的C#语言，以及C++、Visual Basic、JScript甚至一些旧语言，如COBOL。为此，还推出了这些语言的.NET版本，目前还在不断推出更多版本。要获得这些语言的列表，可以访问

<https://msdn.microsoft.com/library/aa292164.aspx>。所有这些语言都可以访问.NET Framework，它们彼此之间还可以通信。C#开发人员可以使用Visual Basic程序员编写的代码，反之亦然。

所有这些提供了意想不到的多样性，这也是.NET Framework具有诱人前景的部分原因。

1.1.1 .NET Framework的内容

.NET Framework主要包含一个庞大的代码库，可以在客户语言（如C#）中通过面向对象编程技术（OOP）来使用这些代码。这个库分为多个不同的模块，这样就可以根据希望得到的结果来选择使用其中的各个部分。例如，一个模块包含Windows应用程序的构件，另一个模块包含网络编程的代码块，还有一个模块包含Web开发的代码块。一些模块还分为更具体的子模块，例如，在Web开发模块中，有用于建立Web服务的子模块。

其目的是，不同操作系统可以根据各自的特性，支持其中的部分或全部模块。例如，智能手机支持所有的核心.NET功能，但不需要某些更高级的模块。

部分.NET Framework库定义了一些基本类型。类型是数据的一种表达方式，指定最基本类型（如32位带符号的整数）有助于使用.NET Framework的各种语言之间进行交互操作，这称为通用类型系统（Common Type System，CTS）。

除提供这个库外，.NET Framework还包含.NET公共语言运行库

（Common Language Runtime, CLR），它负责管理用.NET库开发的所有应用程序的执行。

1.1.2 使用.NET Framework编写应用程序

使用.NET Framework编写应用程序，就是使用.NET代码库编写代码（使用支持Framework的任何一种语言）。本书用VS进行开发，VS是一种强大的集成开发环境，支持C#（以及托管和非托管C++、Visual Basic和其他一些语言）。这个环境的优点是便于把.NET功能集成到代码中。我们创建的代码完全是C#代码，但使用了.NET Framework，并在需要时利用了VS中的其他工具。

为执行C#代码，必须把它们转换为目标操作系统能理解的语言，即本机代码（native code）。这种转换称为编译代码，由编译器执行。但在.NET Framework下，此过程包括两个阶段。

1. CIL和JIT

在编译使用.NET Framework库的代码时，不是立即创建专用于操作系统的本机代码，而是把代码编译为通用中间语言（Common Intermediate Language, CIL）代码，这些代码并非专门用于任何一种操作系统，也非专门用于C#。其他.NET语言（如Visual Basic .NET）也会在第一阶段编译为这种语言。开发C#应用程序时，这个编译步骤由VS完成。

显然，要执行应用程序，必须完成更多工作，这是Just-In-Time（JIT）编译器的任务，它把CIL编译为专用于OS和目标机器结构的本机代码。这样OS才能执行应用程序。这里编译器的名称Just-In-Time反映了CIL代码仅在需要时才编译的事实。这种编译可以在应用程序的运行过程中动态发生，不过开发人员一般不需要关心这个过程。除非要编写性能十分关键的代码，否则知道这个编译过程会在后台自动进行，并不需要人工干预就可以了。

过去，常需要把代码编译为几个应用程序，每个应用程序都用于特定的操作系统和CPU结构。这通常是一种优化形式（例如，为了让代码在AMD芯片组上运行得更快），有时则是非常重要的（例如，使应用程序可以同时工作在Win9x 和WinNT/2000环境下）。现在就没必要了，因为JIT编译器使用CIL代码，而CIL代码是独立于计算机、操作系统和CPU的。目前有几种JIT编译器，每种编译器都用于不同的结构，CIL会使用合适的编译器创建所需的本机代码。

这样，开发人员需要做的工作就比较少了。实际上，可以忽略与系统相关的细节，将注意力集中在代码的功能上就够了。

提示：读者可能遇到过Microsoft Intermediate Language（MSIL）或IL，MSIL是CIL原来的名称，许多开发人员仍沿用这个术语。

2. 程序集

编译应用程序时，所创建的CIL代码存储在一个程序集中。程序集

包括可执行的应用程序文件（这些文件可以直接在Windows上运行，不需要其他程序，其扩展名是.exe）和其他应用程序使用的库（其扩展名是.dll）。

除包含CIL外，程序集还包含元信息（即程序集中包含的数据的信息，也称为元数据）和可选的资源（CIL使用的其他数据，例如，声音文件和图片）。元信息允许程序集是完全自描述的。不需要其他信息就可以使用程序集，也就是说，我们不会遇到没有把需要的数据添加到系统注册表中这样的问题，而在使用其他平台进行开发时这个问题常常出现。

因此，部署应用程序就非常简单了，只需把文件复制到远程计算机上的目录下即可。因为不需要目标系统上的其他信息，所以只需从该目录中运行可执行文件即可（假定安装了.NET CLR）。

当然，不必把运行应用程序需要的所有信息都安装到一个地方。可以编写一些代码来执行多个应用程序所要求的任务。此时，通常把这些可重用的代码放在所有应用程序都可以访问的地方。在.NET Framework中，这个地方是全局程序集缓存（Global Assembly Cache, GAC），把代码放在这个缓存中是很简单的，只需把包含代码的程序集放在包含该缓存的目录中即可。

3. 托管代码

在将代码编译为CIL，再用JIT编译器将它编译为本机代码后，CLR的任务尚未全部完成，还需要管理正在执行的用.NET Framework编写的代码（这个执行代码的阶段通常称为运行时（runtime））。即CLR管理

着应用程序，其方式是管理内存、处理安全性以及允许进行跨语言调试等。相反，不受CLR控制运行的应用程序属于非托管类型，某些语言（如C++）可以用于编写此类应用程序，例如，访问操作系统的底层功能。但是在C#中，只能编写在托管环境下运行的代码。我们将使用CLR的托管功能，让.NET处理与操作系统的任何交互。

4. 垃圾回收

托管代码最重要的一个功能是垃圾回收（garbage collection）。这种.NET方法可确保应用程序不再使用某些内存时，就会完全释放这些内存。在.NET推出以前，这项工作主要由程序员负责，代码中的几个简单错误会把大块内存分配到错误的地方，使这些内存神秘失踪。这通常意味着计算机的速度逐渐减慢，最终导致系统崩溃。

.NET垃圾回收会定期检查计算机内存，从中删除不再需要的内容。执行垃圾回收的时间并不固定，可能一秒钟内会进行数千次的检查，也可能几秒钟才检查一次，不过一定会进行检查。

这里要给程序员一些提示。因为在不可预知的时间执行这项工作，所以在设计应用程序时，必须留意这一点。需要许多内存才能运行的代码应自行完成清理工作，而不是坐等垃圾回收，但这不像听起来那样难。

5. 把它们组合在一起

在继续学习之前，先总结一下上述创建.NET应用程序所需的步骤：

(1) 使用某种.NET兼容语言（如C#）编写应用程序代码，如图1-1所示。

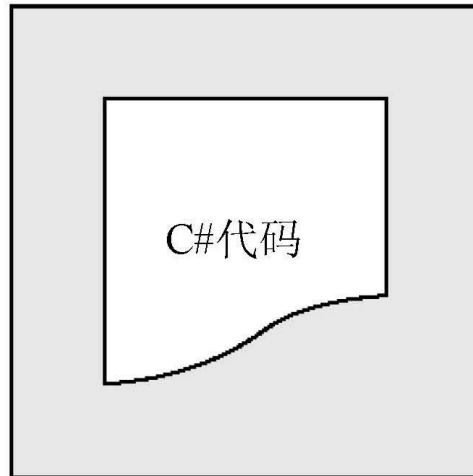


图1-1

(2) 把代码编译为CIL，存储在程序集中，如图1-2所示。

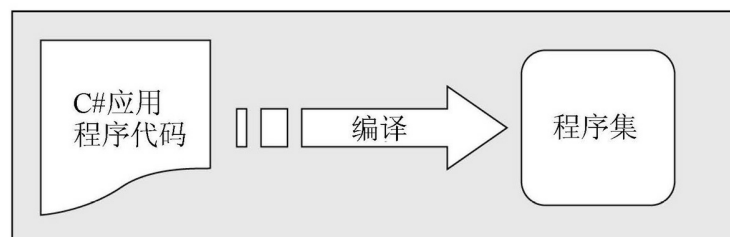


图1-2

(3) 在执行代码时（如果这是一个可执行文件，就自动运行，或者在其他代码使用它时运行），首先必须使用JIT编译器将代码编译为本机代码，如图1-3所示。

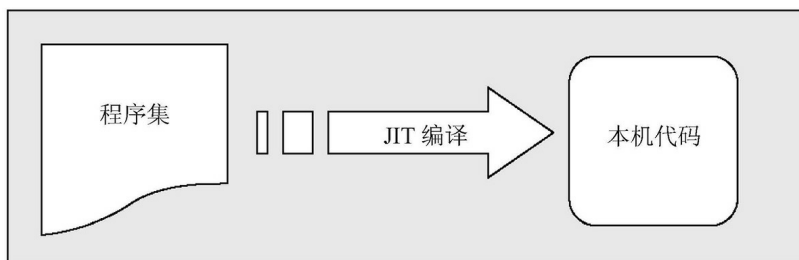


图1-3

（4）在托管的CLR环境下运行本机代码，以及其他应用程序或进程，如图1-4所示。

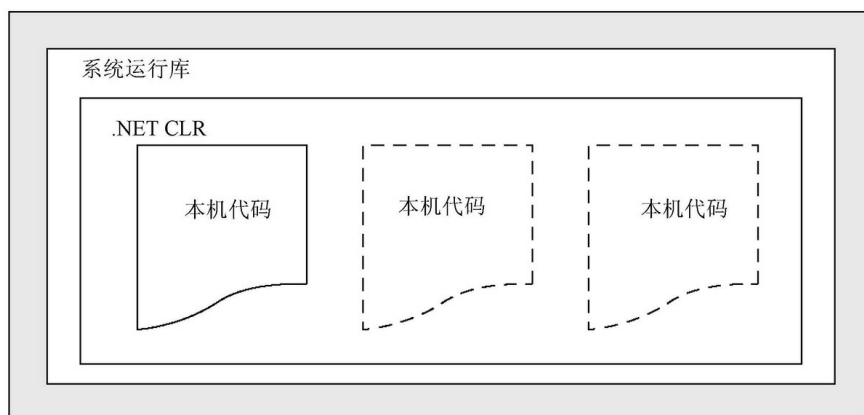


图1-4

6. 链接

在上述过程中还有一点要注意。在第（2）步中编译为CIL的C#代码未必包含在单独文件中，可以把应用程序代码放在多个源代码文件中，再把它们编译到一个程序集中。这个过程称为链接（linking），是非常有用的。原因是处理几个较小的文件比处理一个大文件要简单得多。可以把逻辑上相关的代码分解到一个文件中，以便单独进行处理，

这也更便于在需要时找到特定的代码块，让开发小组把编程工作分解为一些可管理的块，让每个人编写一小块代码，而不会破坏已编写好的代码部分或其他人正在处理的部分。

1.2 C#的含义

如上所述，C#是可用于创建要运行在.NET CLR上的应用程序的语言之一，它从C和C++语言演化而来，是Microsoft专门为使用.NET平台而创建的。C#吸取了以往语言失败的教训，考虑了其他语言的许多优点，并解决了它们存在的问题。

使用C#开发应用程序比使用C++简单，因为其语法更简单。但是，C#是一种强大的语言，在C++中能完成的任务几乎都能利用C#完成。虽然如此，C#中与C++高级功能等价的功能（例如直接访问和处理系统内存），只能在标记为“unsafe”的代码中使用。顾名思义，这个高级编程技术存在潜在威胁，因为它可能覆盖系统中重要的内存块，导致严重后果。因此，本书不讨论这个问题。

C#代码常比C++略长一些。这是因为C#是一种类型安全的语言（与C++不同）。在外行人看来，这表示一旦为某个数据指定了类型，就不能转换为另一个不相关的类型。所以，在类型之间转换时，必须遵守严格的规则。执行相同的任务时，用C#编写的代码通常比用C++编写的代码长。但C#代码更健壮，调试起来也比较简单，.NET始终可以随时跟踪数据的类型。在C#中，不能完成诸如“把4字节的内存分配给这个数据后，我们使其有10个字节长，并把它解释为X”等任务，但这并不是一件坏事。

C#只是用于.NET开发的一种语言，但它是最好的一种语言。C#的优点是，它是唯一彻头彻尾为.NET Framework设计的语言，是在移植到其他操作系统上的.NET版本中使用的主要语言。要使诸如VB.NET的语

言尽可能类似于其以前的语言，且仍遵循CLR，就不能完全支持.NET代码库的某些功能，至少需要不常见的语法。

但C#能使用.NET Framework代码库提供的每种功能。而且，.NET的每个新版本都在C#语言中添加了新功能，满足了开发人员的要求，使之更强大。

1.2.1 用C#能编写什么样的应用程序

如前所述，.NET Framework没有限制应用程序的类型。C#使用的是.NET Framework，所以也没有限制应用程序的类型。这里仅讨论几种常见的应用程序类型。

- **桌面应用程序** 这些应用程序（如Microsoft Office）具有我们很熟悉的Windows外观和操作方式，使用.NET Framework的Windows Presentation Foundation（WPF）模块就可以简便地生成这种应用程序。WPF模块是一个控件库，其中的控件（例如按钮、工具栏和菜单等）可用于建立Windows用户界面（UI）。
- **Windows Store应用程序** 这是Windows 8引入的一类新的应用程序。此类应用程序主要针对触摸设备设计，通常全屏运行，侧重点在于简洁清晰。创建这类应用程序的方式有多种，包括使用WPF。
- **云/Web应用程序** .NET Framework包括一个动态生成Web内容的强大系统——ASP.NET，允许进行个性化和实现安全性等。另外，这些应用程序可以在云中驻留和访问，例如Microsoft Azure平台。
- **Web API** 这是建立REST风格的HTTP服务的理想框架，支持许多客户端，包括移动设备和浏览器。
- **WCF服务** 这是一种灵活创建各种分布式应用程序的方式。使用

WCF服务可以通过局域网或Internet交换几乎各种数据。无论使用什么语言创建WCF服务，也无论WCF服务驻留在什么系统上，都使用一样简单的语法。

这些类型的应用程序也可能需要某种形式的数据库访问，这可以通过.NET Framework的Active Data Objects .NET (ADO.NET) 部分、ADO.NET Entity Framework或C#的LINQ (Language Integrated Query) 功能来实现。也可以使用许多其他资源，例如，创建联网组件、输出图形、执行复杂数学任务的工具。

1.2.2 本书中的C#

本书第 I 部分介绍C#语言的语法和用法，但不过分强调.NET Framework。这是必需的，因为我们不能没有一点儿C#编程基础就使用.NET Framework。首先介绍一些比较简单的内容，把较复杂的面向对象编程 (Object-Oriented Programming, OOP) 主题放在基础知识的后面论述。假定读者没有一点儿编程的知识，这些是首要原则。

学习了基础知识后，本书还将介绍如何开发更复杂、更有用的应用程序。本书第 II 部分将研究基于云的Web应用程序编程，第III部分将讲述数据访问 (对ORM数据库、文件系统和XML数据的访问) 和LINQ，第IV部分将详细讨论桌面和Windows Store应用程序编程。

1.3 Visual Studio 2015

本书使用Visual Studio 2015开发工具进行所有的C#编程，包括简单的命令行应用程序，乃至较复杂的项目类型。VS不是开发C#应用程序必需的开发工具或集成开发环境（IDE），但使用它可以使任务更简单一些。如果愿意的话，可在基本的文本编辑器（如常见的记事本）中处理C#源代码文件，再使用.NET Framework中包含的命令行编译器把代码编译到程序集中。但是，为什么不使用功能完备的IDE呢？

1.3.1 Visual Studio Express 2015产品

除Visual Studio 2015外，Microsoft还提供了几个更简单的开发工具，称为Visual Studio Express或Community 2015产品。可以在<https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs>上免费获得它们。

各种Express产品可以创建所需的几乎所有C#应用程序。在功能上它们都是VS的删节版本，但外观和操作方式是一样的。尽管它们提供了VS的许多功能，但缺少一些重要功能；不过我们仍可以在学习本书的过程中使用它们。

注意： 由于在写作本书时Express版本还不可用，本书使用了Visual Studio 2015的企业版。在写作本书时，有一个称为Visual

Studio Express 2015 for Windows Desktop的Express产品预计在不久后会发布，使用它应该足以学习本书的第I部分。对于本书的剩余部分，使用Visual Studio Express 2015 for Windows 10和Visual Studio Express 2015 for Web应该也是可以的，但是在写作本书的时候我们不能肯定这一点一定成立。

1.3.2 解决方案

在使用VS开发应用程序时，可以通过创建解决方案来完成。在VS术语中，解决方案不仅是一个应用程序，它还包含项目，可以是WPF项目和Cloud/Web应用程序项目等。但是，解决方案可以包含多个项目，这样，即使相关的代码最终在硬盘上的多个位置被编译为多个程序集，也可以把它们组合到一处。

这是非常有用的，因为它可以处理“共享”代码（这些代码放在GAC中），同时，应用程序也使用这段共享代码。在使用唯一的开发环境时，调试代码是非常容易的，因为可以在多个代码块中单步调试指令。

1.4 本章要点

主题	要点
.NET Framework 基础	.NET Framework是Microsoft最新的开发平台，目前的版本是4.6。它包括一个公共类型系统（CTS）和一个公共语言运行库（CLR）。.NET Framework应用程序使用面向对象编程（OOP）的方法论编写，通常包含托管代码。托管代码的内存管理由.NET运行库处理，其中包括垃圾回收
.NET Framework 应用程序	用.NET Framework编写的应用程序首先编译为CIL。在执行应用程序时，JIT把CIL编译为本机代码。应用程序编译后，把不同的部分链接到包含CIL的程序集中
C#基础	C#是包含在.NET Framework中的一种语言，它是已有语言（如C++）的一种演变，可用于编写任意应用程序，包括Web应用程序和桌面应用程序
集成开发环境（IDE）	可在Visual Studio 2015中用C#编写任意类型的.NET应用程序，还可以在免费的但功能稍弱的Express产品系列中用C#创建.NET应用程序。这两种IDE都使用解决方案，解决方案可以包含多个项目

第2章 编写C#程序

本章内容：

- Visual Studio 2015的基础知识
- 编写简单的控制台应用程序
- 编写简单的桌面应用程序

本章源代码下载：

本章源代码的下载网址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 2 Code后，可以找到与本章示例对应的单独文件。

第1章占用一定的篇幅讨论了C#是什么，以及它是如何适应.NET Framework的，现在就该编写一些代码了。本书主要使用Visual Studio 2015（VS 2015），所以首先介绍这个开发环境的一些基础知识。

VS是一个庞大的复杂产品，可能会使初学者望而生畏，但使用它创建简单的应用程序是非常容易的。在本章开始使用VS时，不需要了解许多知识，就可以编写C#代码。本书的后面将介绍VS能执行的更复杂操作，现在仅介绍基础知识。

介绍完IDE后，将创建两个简单应用程序。现在不要过多地考虑代码，只要应用程序可以运行即可。在这些早期的示例中熟悉了应用程序的创建过程，不久后就会适应这个过程了。

本章将学习创建两种基本的应用程序类型：控制台应用程序和桌面应用程序。

下面要创建的第一个应用程序是一个简单的控制台应用程序。控制台应用程序没有使用图形化的Windows环境，所以不需要考虑按钮、菜单、用鼠标指针进行交互等，而是在命令行窗口中运行应用程序，用更简单的方式与其交互。

第二个应用程序是使用Windows Presentation Foundation（WPF）创建的一个桌面应用程序，其外观和操作方式对Windows用户来说会非常熟悉，而且该应用程序创建起来并不费力。但所需代码的语法比较复杂，尽管在许多情况下，并不需要考虑细节。

本书的第III部分和第IV部分也使用这两种应用程序类型，但开始时主要讨论控制台应用程序。在学习C#语言时，不需要了解桌面应用程序的其他灵活性能。控制台应用程序的简单性可以让我们集中精力学习语法，而不必考虑应用程序的外观和操作方式。

2.1 Visual Studio 2015开发环境

在首次加载VS时，会立即显示选项Sign in to Visual Studio using your Microsoft Account（用Microsoft账户注册Visual Studio）。注册后，Visual Studio设置就会在设备上同步，在多个工作站上使用IDE时，就不必配置它。如果没有Microsoft账户，可以创建一个，再使用它注册。如果不希望注册，就单击Not now, maybe later链接，继续Visual Studio的初始配置。有时建议注册，获得一个开发人员许可证。

如果是首次运行VS，则屏幕上会显示一个首选项列表。如果用户使用过这个开发环境的旧版本，则可以在这里做出选择，这些选择会影响到很多方面，例如窗口的布局、控制台窗口运行的方式等。所以应选择Visual C# Development Settings，否则会发现一些地方和本书的描述不一样。注意，可用选项会随着安装VS时选择的选项而变化，但只要选择安装C#，这个选项就是可用的。

如果不是第一次运行VS，但以前选择了另一个选项，也不必惊慌。为把设置重置为Visual C# Development Settings，只需导入它们即可。为此，单击Tools菜单上的Import and Export Settings选项，再选中Reset all settings选项，如图2-1所示。

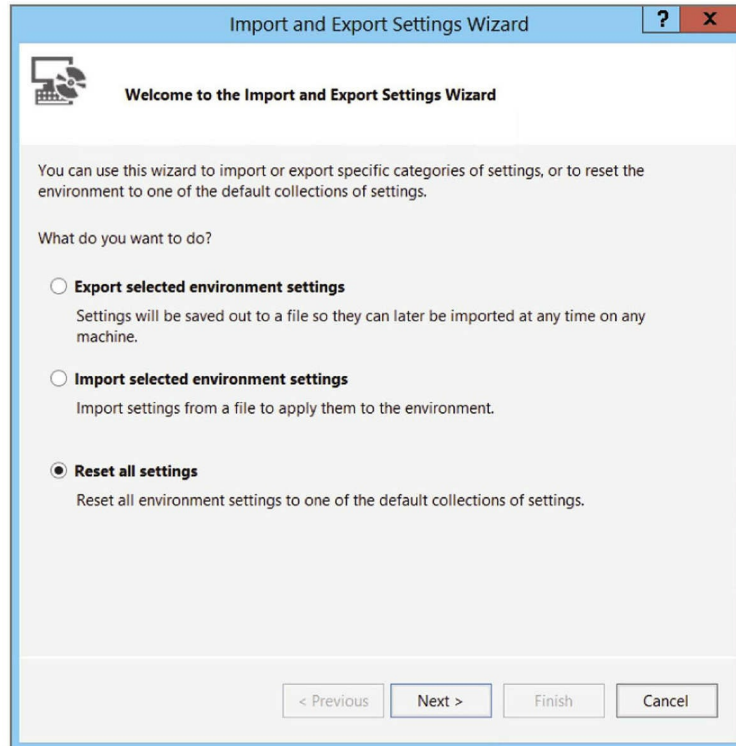


图2-1

单击Next按钮，选择是否要在继续之前保存已有的设置。如果对设置进行了定制，就保存设置，否则选择No按钮，再次单击Next按钮。在下个对话框中，选择Visual C#选项，如图2-2所示。可用的选项可能会变化。

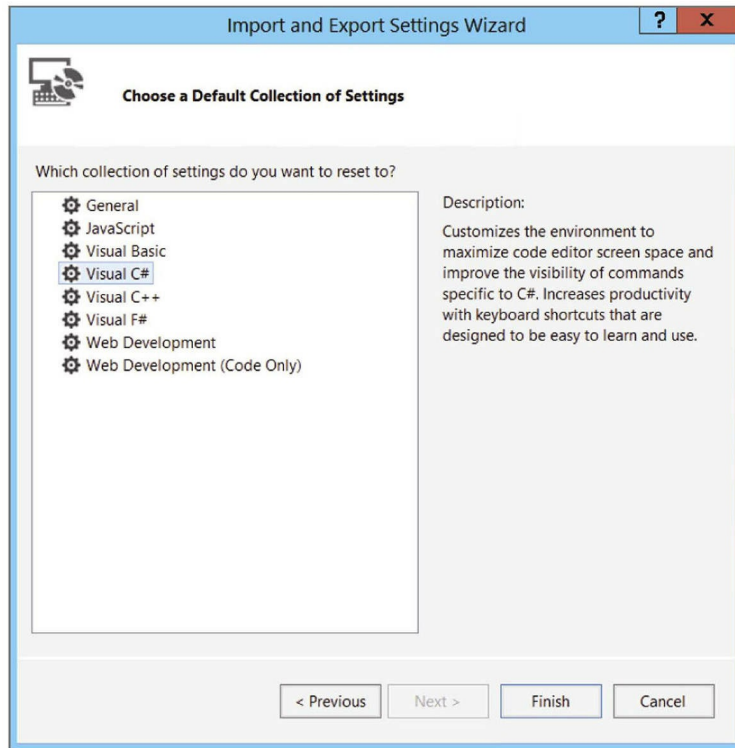


图2-2

最后单击Finish按钮，应用设置。

VS环境布局是完全可定制的，但默认设置很适合我们。在C# Developer Settings设置下，其布局如图2-3所示。

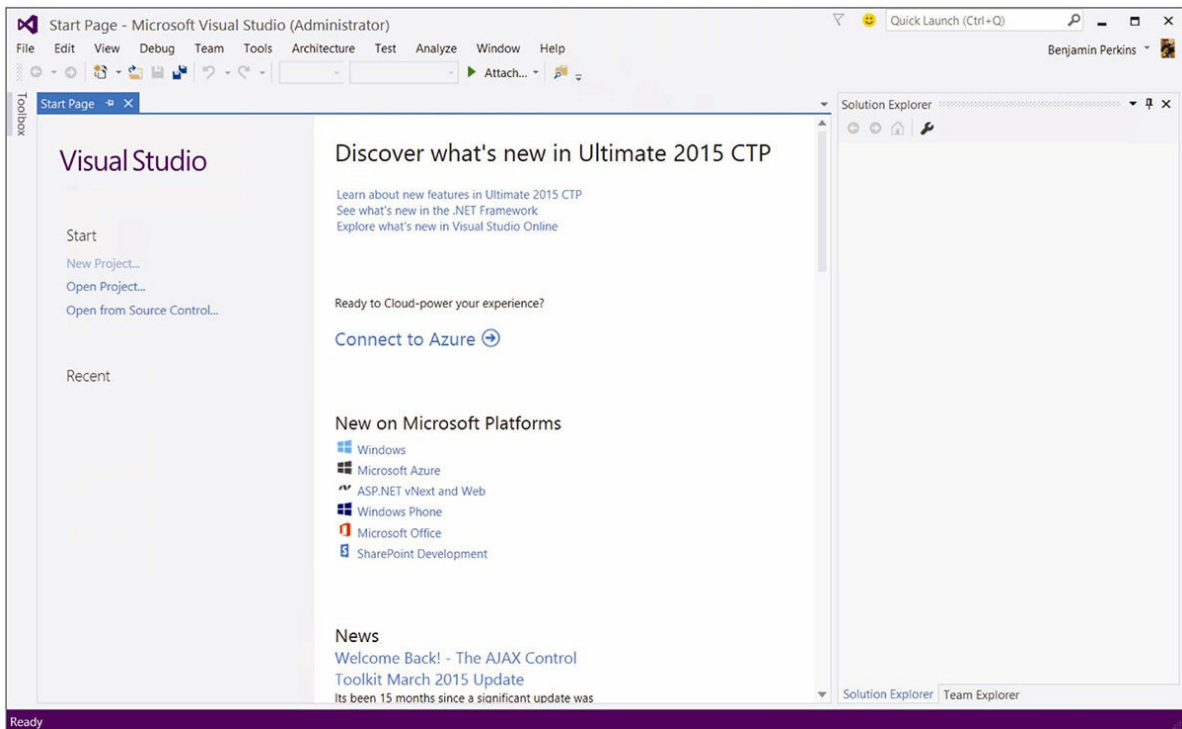


图2-3

所有代码都显示在主窗口中。在VS启动时，主窗口会默认显示一个提供帮助信息的Start Page。主窗口可以包含许多文档，每个文档都有一个选项卡，单击文件名，就可以在文件之间切换。这个窗口也具有其他功能：它可以显示为项目设计的GUI、纯文本文件、HTML以及各种内置于VS的工具。本书将陆续介绍它们。

在主窗口的上面，有工具栏和VS菜单。这里有几个不同的工具栏，其功能包括：保存和加载文件、生成和运行项目，以及调试控件等。在需要使用这些工具栏时将会讨论它们。

下面简要描述VS的最常用功能：

- 单击Toolbox选项卡时，就会显示Toolbox工具栏，它提供了桌面应用程序的用户界面构件等条目。另一个选项卡Server Explorer也可

以在这里显示（通过View|Server Explorer菜单项选择它），它包含其他许多功能，例如Azure订阅细节、访问数据源、服务器设置和服务等。

- **Solution Explorer**窗口显示当前加载的解决方案的信息。如上一章所述，解决方案是一个VS术语，表示一个或多个项目及其配置。
Solution Explorer窗口显示了解决方案中项目的各种视图，例如项目中包含了哪些文件，这些文件中又包含了什么内容。
- **Team Explorer**窗口显示了关于当前的**Team Foundation Server**或**Team Foundation Service**连接的信息，可用于使用源代码管理、bug跟踪、自动生成等功能。但是，这是一个高级主题，本书不予介绍。
- **Solution Explorer**窗口之下可以显示**Properties**窗口，该窗口没有显示在图2-3中。稍后会看到这个窗口，因为它只在处理项目时才出现（也可以使用View|Properties Window菜单项切换它）。这个窗口提供了更详细的项目内容视图，允许另外配置单独元素。例如，使用这个窗口可以改变桌面应用程序中按钮的外观。
- 另一个非常重要的窗口也未出现在图2-3中：**Error List**窗口。可以使用View|Error List菜单项打开这个窗口，它显示了错误、警告和其他与项目有关的信息。这个窗口会持续不断地更新，但其中一些信息只有在编译项目时才出现。

这似乎需要理解很多东西，但不必担心，过不了多久就习惯了。下面首先建立第一个示例项目，它将使用上面介绍的许多VS元素。

注意： VS还可以显示许多其他窗口，它们都包含许多信息，有许多功能。其中一些窗口与上面提及的窗口共享屏幕空间，可以使用

选项卡切换它们或把它们停靠在其他位置。如果有多个显示器，甚至可以分离它们，把它们放到其他显示器上显示。本书的后面会介绍其中的许多窗口，在读者自己深入探索VS环境时，可能还会发现更多窗口。

2.2 控制台应用程序

本书将频繁使用控制台应用程序，特别是开始时要使用这类应用程序，所以下面分步演示如何创建一个简单的控制台应用程序。

试一试：创建一个简单的控制台应用程序：

ConsoleApplication1\Program.cs

（1）选择File|New|Project菜单项，创建一个新的控制台应用程序项目，如图2-4所示。

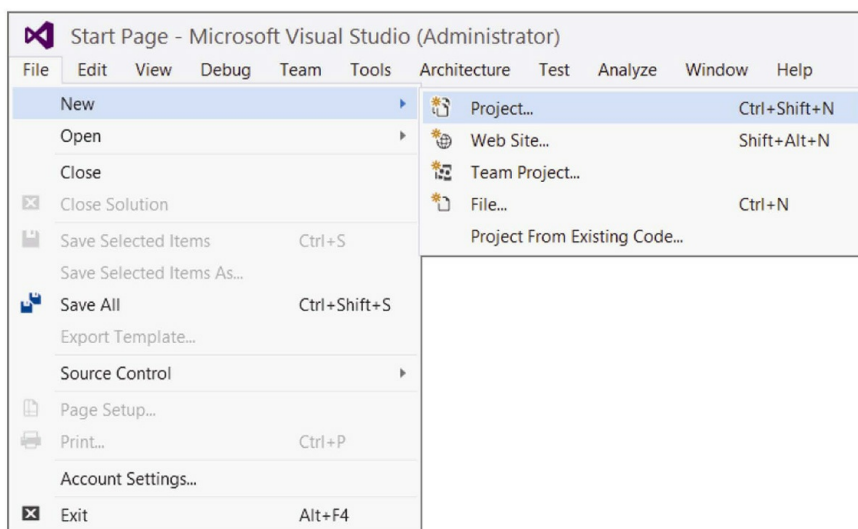


图2-4

（2）在显示窗口的左侧选择Visual C#节点，在中间窗格中选择

Console Application项目类型，如图2-5所示。把Location文本框改为C:\BegVCSharp\Chapter02（如果该目录不存在，会自动创建）。Name文本框中的默认文本（ConsoleApplication1）和其他设置不变，参见图2-5。

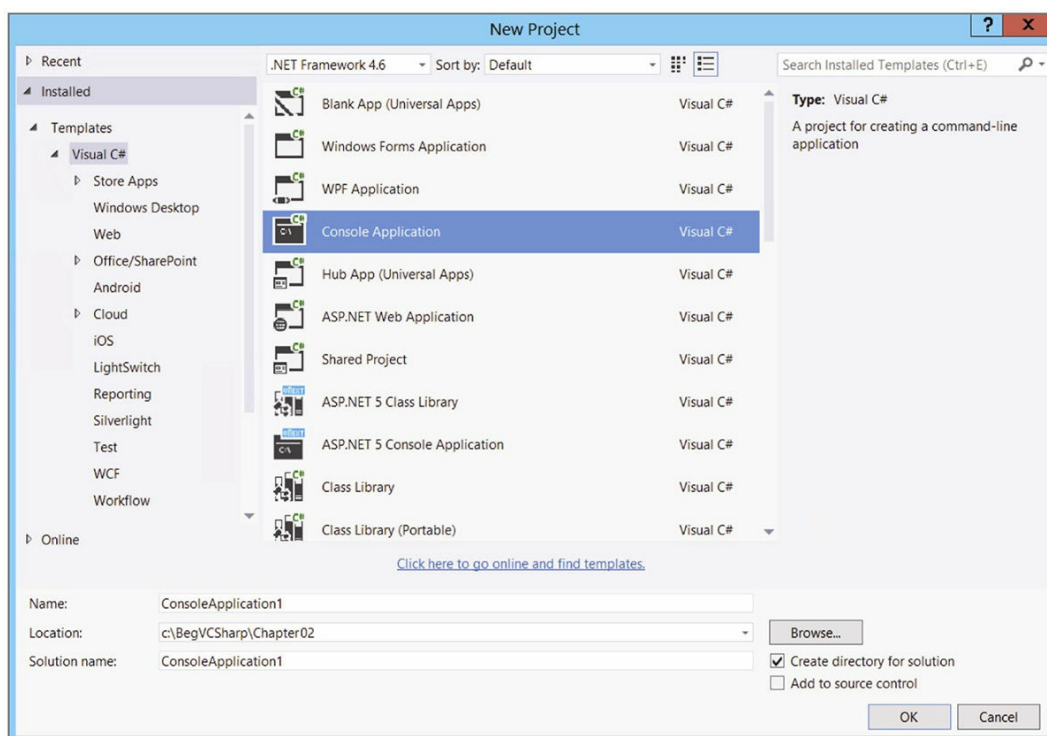


图2-5

（3）单击OK按钮。

（4）初始化项目后，在主窗口显示的文件中添加如下代码行：

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
```

```

{
    // Output text to the screen.

    Console.WriteLine("The first app in Beginning Visual C# 2015!");

    Console.ReadKey();
}
}
}

```

(5) 选择Debug|Start Debugging菜单项。稍后将看到如图2-6所示的结果。

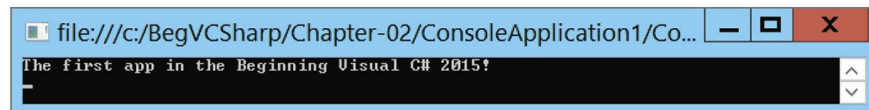


图2-6

(6) 按下任意键，退出应用程序（可能需要首先单击控制台窗口，以激活它）。只有像本章前面描述的那样应用了Visual C# Developer Settings，才会显示图2-6所示内容。例如，若应用了Visual

Basic Developer Settings，就会显示一个空的控制台窗口，应用程序的输出结果显示在Immediate窗口中。这种情况下，`Console.ReadKey()`代码也会失败，显示一个错误。如果遇到这个问题，本书中所有示例的最佳解决方案是应用Visual C# Developer Settings，这样读者看到的结果才会与书中显示的相同。

示例说明

现在不仔细研究这个项目中使用的代码，而关心如何使用开发工具来启动和运行代码。显然，VS自动完成了许多工作，简化了编译和执行代码的过程。执行这些简单的步骤还有多种方式。例如，创建一个新项目可以像前面那样使用菜单项，也可以按下`Ctrl+Shift+N`组合键，还可以单击工具栏上的相应图标。

同样，也可以采用多种方式编译和执行代码。上面使用的方法是选择Debug|Start Debugging菜单项，也可以按下快捷键（F5），或者使用工具栏中的图标。使用Debug|Start Without Debugging菜单项（也可以按下`Ctrl+F5`组合键）还可以采用非调试模式运行代码，使用Build|Build Solution菜单项或F6快捷键可以编译项目但不运行它（打开或关闭调试功能）。注意，执行项目但不调试，或者使用工具栏中的图标生成项目，只是这些图标在默认情况下没有显示在工具栏中。编译好代码后，在Windows资源管理器中运行生成的.exe文件，就可以执行代码。也可以在命令提示窗口中执行，为此，应打开一个命令提示窗口，把目录改为

C:\BegVCSharp\Chapter02\ConsoleApplication1\ConsoleApplication1\bin\Debug
键入ConsoleApplication1，并按下回车键。

注意： 在以后的示例中，我们仅说明“创建一个新的控制台项目”或“执行代码”，用户可以选择自己喜欢的方式执行这些步骤。除非特别声明，否则所有的代码都应在启用调试的情况下运行。另外，本书中的“启动”、“执行”和“运行”等术语的含义是相同的，示例后面的讨论总是假定已经退出了示例中的应用程序。

控制台应用程序会在执行完毕后立即终止，如果直接通过IDE运行它们，就无法看到运行结果。为解决上例中的这个问题，使用

```
Console.ReadKey();
```

告诉代码在结束前等待按键。后面的示例将多次使用这种技术。前面创建了一个项目，现在详细讨论开发环境中的各个组成部分。

2.2.1 Solution Explorer窗口

Solution Explorer窗口默认位于屏幕右上角。与其他窗口一样，可把它移到任何位置，或者单击其图钉图标将它设为自动隐藏。Solution Explorer窗口与另一个有用的窗口Class View位于相同的位置，使用View|Class View菜单项就可以显示Class View窗口。图2-7显示了展开所有节点的这两个窗口（在窗口停靠时，单击窗口底部的选项卡，就可以切换它们）。

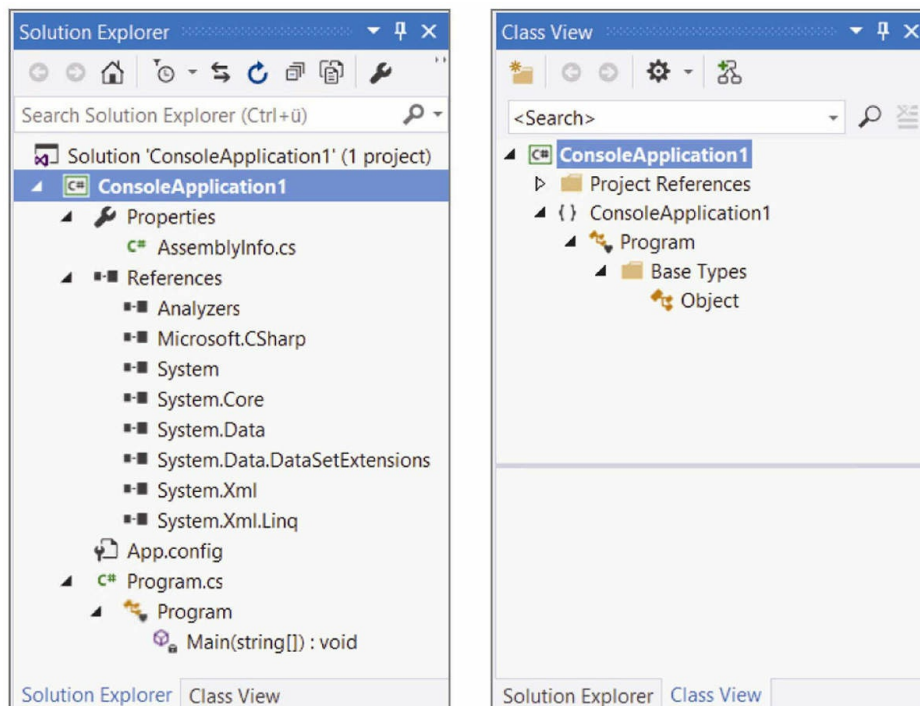


图2-7

Solution Explorer窗口显示了组成ConsoleApplication1项目的文件，包括我们在其中添加代码的文件Program.cs、另一个代码文件AssemblyInfo.cs和多个引用。

注意： 所有C#代码文件都使用.cs文件扩展名。

此时不需要考虑AssemblyInfo.cs文件，它包含项目中目前我们不必关心的其他信息。

使用这个窗口可以改变主窗口中显示的代码，方法是双击.cs文件，或右击这些文件并选择View Code，或选中它们，单击窗口顶部的工具

栏按钮。还可以对这些文件执行其他操作，例如，重命名它们，或从项目中删除它们等。在该窗口中还可以显示其他类型的文件，例如，项目资源（资源是项目使用的文件，这些文件可能不是C#文件，如位图图像和声音文件等）。可以通过同一界面处理它们。

展开代码项（例如**Program.cs**）可以查看其中包含的内容。这个代码结构概览是一个很有帮助的工具，可用来直接定位到代码文件中的特定部分，而不必打开该代码文件并滚动到想要处理的部分。

References项包含项目中使用的一个.NET库列表，这个列表在后面介绍，因为标准引用很适于初学者使用。**Class View**窗口显示了项目的另一种视图，可以用于查看刚才创建的代码结构。本书后面将介绍代码结构，现在使用**Solution Explorer**窗口就足够了。单击这些窗口中的文件或其他图标，**Properties**窗口的内容就会发生相应变化，如图2-8所示。

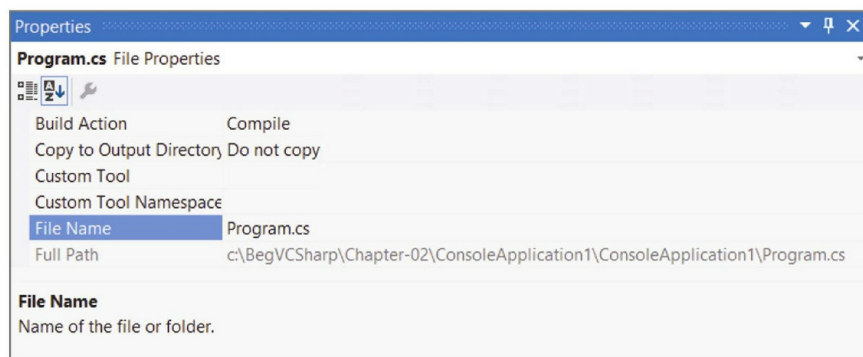


图2-8

2.2.2 Properties窗口

使用**View|Properties Window**菜单项就可以打开**Properties**窗口。这个窗口显示了在其上面的窗口中所选的项的其他信息。例如，选择项目中

的Program.cs文件，就会显示如图2-8所示的窗口。这个窗口还显示了其他选中项的信息，例如用户界面组件（参见本章的2.3节“桌面应用程序”）。

通常在Properties窗口中对项目的改变会直接影响代码，添加代码行，或改变文件中的内容。对于一些项目来说，通过这个窗口来操作与手动修改代码所用的时间是相同的。

2.2.3 Error List窗口

当前Error List窗口（View|Error List）没有显示什么有趣的信息，这是因为应用程序没有错误。但这的确是一个非常有用的窗口。下面进行测试，从上一节添加的代码中删除某一行的分号。稍后将看到如图2-9所示的结果。

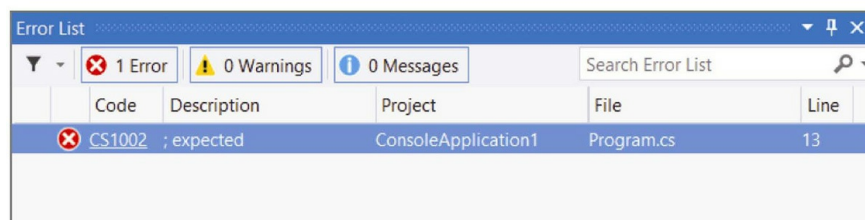


图2-9

这次项目不会编译。

注意： 第3章介绍C#语法后，你就会明白大多数代码行的末尾必须有一个分号。

这个窗口有助于根除代码中的错误，因为它会跟踪我们的工作，编译项目。如果双击该窗口中显示的错误，光标就会跳到源代码中出错的地方（如果包含错误的源文件没有打开，它将被打开），这样就可以快速更正错误。代码中有错误的一行会出现红色的波浪线，以便我们快速浏览源代码，找出错误。

注意错误位置用一个行号来指定。默认情况下，行号不会显示在VS文本编辑器中，但其实有必要显示它。为此，需要单击Tools|Options菜单项，选中Options对话框中的Line numbers复选框。该复选框位于Text Editor|All Languages|General类别中。

也可以在这个对话框中与各个语言对应的设置页面中针对具体语言单独修改此设置。这个对话框中还包含其他许多有用的选项，本书将使用其中几个选项。

2.3 桌面应用程序

通常，在演示代码时，将其当作桌面Windows应用程序的一部分来运行，要比通过控制台窗口或命令提示符来运行更便于说明。下面用用户界面构件来组合一个用户界面。

下面的示例介绍建立用户界面的基础知识，说明如何启动和运行桌面应用程序，但并不详细讨论应用程序实际完成的工作。Microsoft推荐使用WPF技术创建桌面应用程序，所以本例中使用了WPF。本书后面会详细研究桌面应用程序，以及WPF到底是什么，它到底可以做些什么。

试一试：创建一个简单的**Windows**应用程序：

WpfApplication1\MainWindow.xaml和

WpfApplication1\MainWindow.xaml.cs

(1) 在与之前相同的位置（C:\BegVCSharp\Chapter02）创建一个类型为WPF Application的新项目，其默认名称是WpfApplication1。如果第一个项目仍处于打开状态，就应选择Create New Solution选项来启动一个新解决方案，这些设置如图2-10所示。

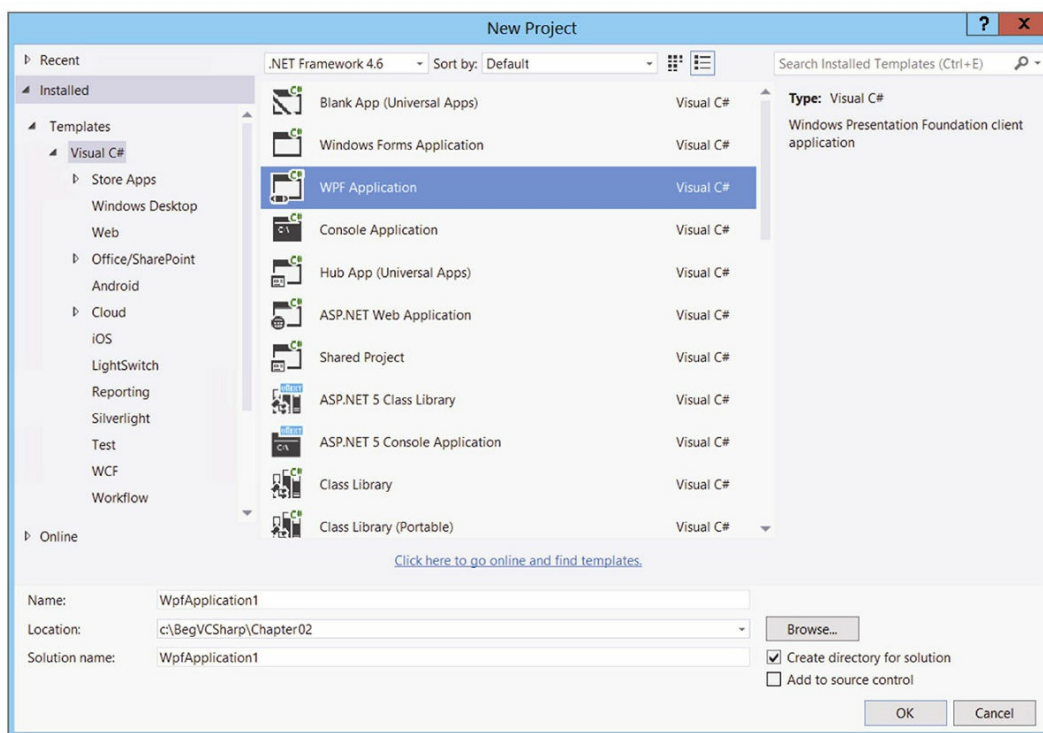


图2-10

(2) 单击OK按钮，创建项目后，应该会看到一个新的分成两个窗格的选项卡。上面的窗格显示了一个空窗口，称为MainWindow，下面的窗格显示了一些文本。这些文本实际上就是用来生成窗口的代码，在修改UI时，会看到这些文本也发生了变化。

(3) 单击屏幕左上方的Toolbox选项卡，然后双击Common WPF Controls区域中的Button，在窗口中添加一个按钮。

(4) 双击刚才添加到窗口中的按钮。

(5) 现在应显示MainWindow.xaml.cs中的C#代码。执行如下修改（为简短起见，这里只显示了文件中的部分代码）：

```
private void Button_Click(object sender, RoutedEventArgs e)
```

```
{  
    MessageBox.Show("The first desktop app in the book!")  
  
}
```

(6) 运行应用程序。

(7) 单击显示出来的按钮，打开一个消息对话框，如图2-11所示。

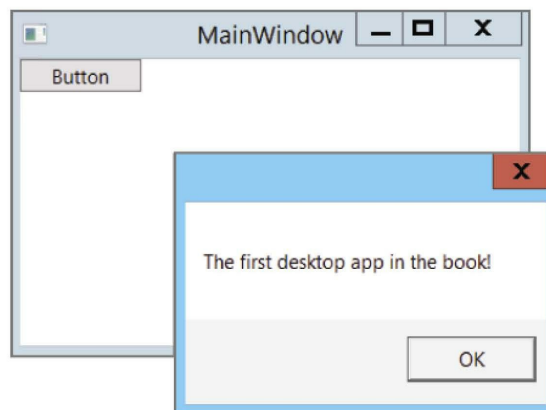


图2-11

(8) 单击OK。像每个标准桌面应用程序那样，单击右上角的X图标，退出应用程序。

示例说明

IDE又一次自动完成了许多工作，使我们不费吹灰之力就能完成一

个实用的桌面应用程序的创建。刚才创建的应用程序与其他窗口的行为方式相同——可以移动、重新设置其大小、最小化等。我们不必编写任何代码来实现这种功能。我们添加的按钮也是这样。双击按钮，IDE就知道我们想添加一些代码，当运行应用程序时，用户单击该按钮，就执行我们已经编写好的代码。只要提供了这段代码，就可以得到按钮单击的所有功能。

当然，桌面应用程序不仅限于带有按钮的普通窗口。如果看看从中选择Button选项的工具箱，就会看到一整套用户界面构件（称为控件），其中一些用户可能很熟悉。本书在其他地方将使用其中的大多数用户界面构件，它们使用起来都非常简单，可以节省许多时间和精力。

应用程序的代码在MainWindow.xaml.cs中，看起来并不比上一节提供的代码复杂多少，Solution Explorer窗口中其他文件的代码也不太复杂。MainWindow.xaml中的代码（可在添加按钮的拆分窗格视图中看到）看上去也很简单。

这是一段XAML代码。XAML是在WPF应用程序中定义用户界面的语言。

下面仔细分析一下在窗口中添加的按钮。在MainWindow.xaml的顶部窗格中，单击按钮一次选中它。此时屏幕右下角的Properties窗口显示了按钮控件的属性（控件也有属性，就像上一个示例中的文件一样）。确保应用程序当前没有运行，然后向下滚动到Content属性，该属性现在被设为Button。将它设为Click Me，如图2-12所示。

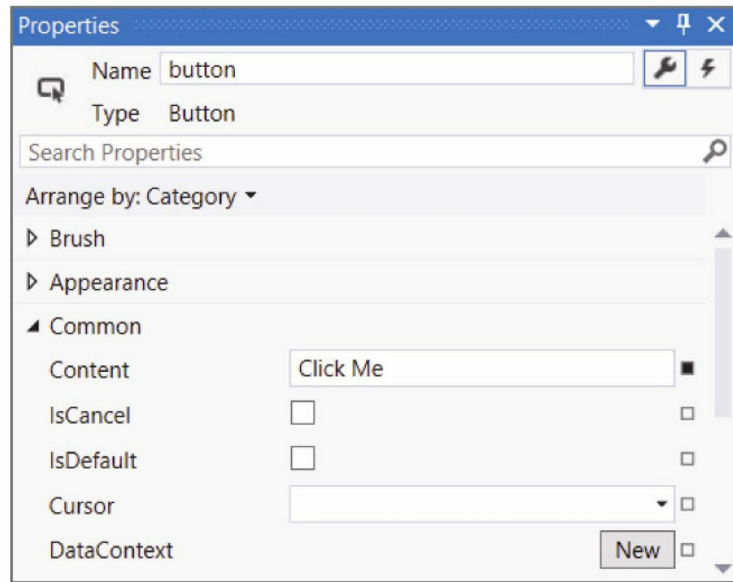


图2-12

设计器中按钮上的文本以及XAML代码也会反映这种变化，如图2-13所示。

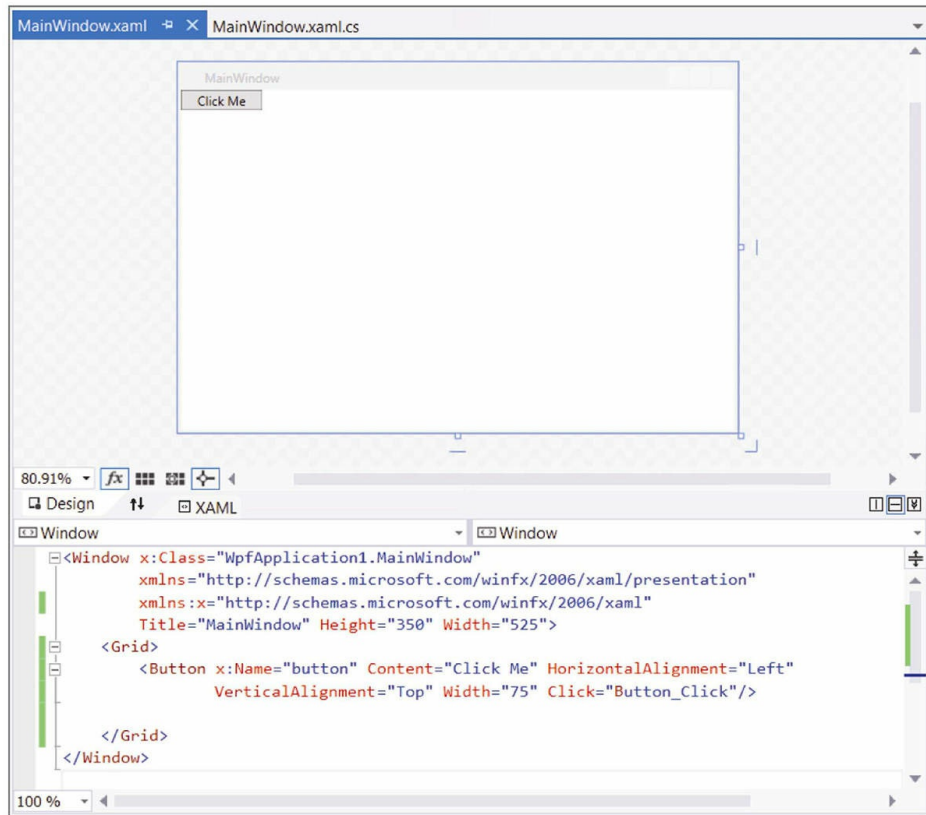


图2-13

这个按钮具有许多属性，从按钮颜色和大小简单格式，到某些模糊设置（如数据绑定设置，它可以建立与数据的联系），应有尽有。如上例所述，改变属性通常会直接改变代码，这也不例外，从XAML代码的改变中可以看到这一点。但如果切换回MainWindow.xaml.cs的代码视图，是看不到代码发生变化的。这是因为WPF应用程序能够保持应用程序的设计（如按钮上的文本）与功能（如单击按钮后发生的操作）的分离。

注意： 也可以使用Windows Forms来创建桌面应用程序。但WPF是一种更新的技术，能够以更灵活、更强大的方式创建桌面应用程序

序，而且其目的就是取代Windows Forms，所以本书中不讨论Windows Forms。

2.4 本章要点

主题	要点
Visual Studio 2015 设置	本书需要在第一次运行VS时选择C# Development Settings选项，或者重置它们
控制台应用程序	控制台应用程序是简单的命令行应用程序，本书主要用它演示技术。在VS中创建新项目时，使用Console Application模板就会创建新的控制台应用程序。要在调试模式下运行项目，可使用Debug Start Debugging菜单项或者按下F5功能键
IDE窗口	项目内容显示在Solution Explorer窗口中。选中项的属性显示在Properties窗口中。错误显示在Error List窗口中
桌面应用程序	桌面应用程序具备标准Windows应用程序的外观和操作系统方式，包括最大化、最小化和关闭应用程序等大家熟悉的图标。它们是在New Project对话框中用WPF Application模板创建的

第3章 变量和表达式

本章内容：

- C#的基本语法
- 变量及其用法
- 表达式及其用法

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 3 Code后，可以找到与本章示例对应的单独文件。

要高效地使用C#，就一定要理解创建计算机程序时真正需要做些什么。计算机程序最基本的描述也许是一系列处理数据的操作，即使对于最复杂的示例，例如Microsoft Office套装软件之类的大型多功能Windows应用程序，这个论述也正确无误。应用程序的用户虽然看不到它们，但这些操作总是在后台进行。

为进一步解释它，考虑一下计算机的显示单元。我们常常比较熟悉屏幕上的内容，很难不把它设想为“移动的图片”。但实际上，我们看到的仅是一些数据的显示结果，其最初的形式是存储在计算机内存中的0

和1数据流。因此我们在屏幕上执行的任何操作，无论是移动鼠标指针、单击图标或在字处理器中输入文本，都会改变内存中的数据。

当然，还可以利用一些较简单的情形来说明这一点。如果使用计算器应用程序，就要提供数字，对这些数字执行操作，就像用纸和笔计算数字一样，但使用程序会快得多。

如果计算机程序是对数据执行操作，则说明我们需要以某种方式来存储数据，需要某些方法来处理它们。这两种功能是由变量和表达式提供的，本章将探究它们的一般含义和具体含义。

在开始之前，应该首先了解一下C#编程的基本语法，因为我们需要一个环境来学习使用C#语言中的变量和表达式。

3.1 C#的基本语法

C#代码的外观和操作方式与C++和Java非常类似。初看起来，其语法可能比较混乱，不像某些语言那样与书面英语十分接近。但实际上，在C#编程中，使用这种风格是很合理的，而且不用花太多力气就可以编写出便于阅读的代码。

与其他语言（如Python）的编译器不同，C#编译器不考虑代码中的空格、回车符或制表符（这些字符统称为空白字符）。这样格式化代码时就有很大的自由度，但遵循某些规则将有助于提高代码的可读性。

C#代码由一系列语句组成，每条语句都用一个分号结束。因为空白被忽略，所以一行可以有多条语句，但从可读性的角度看，通常在分号的后面加上回车符，不在一行中放置多条语句。但一条语句放在多行是可以的（也比较常见）。

C#是一种块结构的语言，所有语句都是代码块的一部分。这些块用花括号来界定（“{”和“}”），代码块可以包含任意多行语句，或者根本不包含语句。注意花括号字符不需要附带分号。

例如，简单的C#代码块如下所示：

```
{  
    <code line 1, statement 1>;  
    <code line 2, statement 2>  
    <code line 3, statement 2>;
```



```
}
```

其中<code line x , statement y >部分并非真正的C#代码，而是用这个文本作为C#语句的占位符。在这段代码中，第2、第3行代码是同一条语句的一部分，因为在第2行的末尾没有分号。缩进第3行代码，就更容易确定这是第二行代码的继续。

下面的简单示例还使用了缩进格式，提高了C#代码的可读性。这是标准做法，实际上在默认情况下VS会自动缩进代码。一般情况下，每个代码块都有自己的缩进级别，即它向右缩进了多少。代码块可以互相嵌套（即块中可以包含其他块），而被嵌套的块要缩进得多一些。

```
{  
    <code line 1>;  
    {  
        <code line 2>;  
        <code line 3>;  
    }  
    <code line 4>;  
}
```

前面代码行的续行通常也要缩进得多一些，如上面第一个示例中的第3行代码所示。

注意： 在能通过Tools|Options访问的VS Options对话框中，显示了VS用于格式化代码的规则。在Text Editor|C#|Formatting节点的子类别下，包含了其中很多规则。此处的大多数设置都反映了还没有讲述

的C#部分，但如果以后要修改设置，以更适合自己的个性化样式，就可以回过头来看看这些设置。在本书中，为简洁起见，所有代码段都使用默认设置来格式化。

当然，这种样式并不是强制的。但如果不使用它，读者在阅读本书时会很快陷入迷茫之中。

在C#代码中，另一种常见的语句是注释。注释并非严格意义上的C#代码，但代码最好有注释。注释的作用不言自明：给代码添加描述性文本（用英语、法语、德语、外蒙古语等），编译器会忽略这些内容。在开始处理冗长的代码段时，注释可用于为正在进行的工作添加提示，例如“这行代码要求用户输入一个数字”，或“这段代码由Bob编写”。

C#添加注释的方式有两种。可以在注释的开头和结尾放置标记，也可以使用一个标记，其含义是“这行代码的其余部分是注释”。在C#编译器忽略回车符的规则中，后者是一个例外，但这是一种特殊情况。

要使用第一种方式标记注释，可在注释开头加上/*字符，在末尾加上*/字符。这些注释符号可以在单独一行上，也可以在不同的行上，注释符号之间的所有内容都是注释。注释中唯一不能输入的是*/，因为它会被看成注释结束标记。所以下面的语句是正确的：

```
/* This is a comment */  
/* And so. . .  
    . . . is this! */
```

但以下语句会产生错误：

```
/* Comments often end with "*/" characters */
```

注释结束符号后的内容（"*/"后面的字符）会被当作C#代码，因此产生错误。

另一种添加注释的方式是用//开始一个注释，在其后可以编写任何内容，只要这些内容在一行上即可。下面的语句是正确的：

```
// This is a different sort of comment.
```

但下面的语句会失败，因为第二行代码会被解释为C#代码：

```
// So is this,  
    but this bit isn't.
```

这类注释可用于语句的说明，因为它们都放在一行上：

```
<A statement>;           // Explanation of statement
```

前面讲过，有两种给C#代码添加注释的方式。但在C#中，还有第三类注释，严格地说，这是//语法的扩展。它们都是单行注释，用三个/符号来开头，而不是两个。

```
/// A special comment
```

正常情况下，编译器会忽略它们，就像其他注释一样，但可以通过配置VS，在编译项目时，提取这些注释后面的文本，创建一个特殊格式的文本文件，该文件可用于创建文档。为了创建文档，注释必须遵循XML文档的规则，详见

<https://msdn.microsoft.com/library/aa288481.aspx>。本书不讨论这个主题，

但这是很值得探讨的内容，如果读者有时间，建议学习掌握。

特别要注意的一点是，C#代码是区分大小写的。与其他语言不同，必须使用正确的大小写形式输入代码，因为简单地用大写字母代替小写字母会中断项目的编译。看看下面这行代码，它曾在第2章中使用：

```
Console.WriteLine("The first app in Beginning C# Programming!")
```

C#编译器能理解这行代码，因为Console.WriteLine()命令的大小写形式是正确的。但是，下面的语句都不能工作：

```
console.WriteLine("The first app in Beginning C# Programming!")  
CONSOLE.WRITELINE("The first app in Beginning C# Programming!")  
Console.Writeline("The first app in Beginning C# Programming!")
```

这里使用的大小写形式是错误的，所以C#编译器不知道我们要做什么。幸好，VS在代码的键入方面提供了许多帮助，在大多数情况下，它都知道我们要做什么。在键入代码的过程中，VS会推荐用户可能要使用的命令，并尽可能纠正大小写问题。

3.2 C#控制台应用程序的基本结构

下面看看第2章的控制台应用程序示例（ConsoleApplication1），并研究一下它的结构。其代码如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Output text to the screen.
            Console.WriteLine("The first app in Beginning C# Program");
            Console.ReadKey();
        }
    }
}
```

立即就可以看出，上一节讨论的所有语法元素这里都有。其中有分

号、花括号、注释和适当的缩进。

目前看来，这段代码中最重要的部分如下所示：

```
static void Main(string[] args)
{
    // Output text to the screen.
    Console.WriteLine("The first app in Beginning C# Programmin
    Console.ReadKey());
}
```

当运行控制台应用程序时，就会执行这段代码，更确切地讲，是运行花括号中的代码块。如前所述，注释行不做任何事情，包含它们只为了保持代码的清晰。其他两行代码在控制台窗口中输出一些文本，并等待一个响应。但目前我们还不需要关心它的具体机制。

这里要注意一下如何实现第2章介绍的代码大纲功能（虽然在第2章中介绍的是Windows应用程序的代码大纲功能），因为它是一个非常有用的特性。要实现该功能，需要使用`#region`和`#endregion`关键字来定义可以展开和折叠的代码区域的开头和结尾。例如，可以修改针对ConsoleApplication1生成的代码，如下所示：

#region Using directives

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
#endregion
```

这样就可以把这些代码行折叠为一行，以后要查看其细节时，可以再次展开它。这里包含的**using**语句和其下的**namespace**语句将在本章后面予以解释。

注意： 以**#**开头的任意关键字实际上是一个预处理指令，严格地说并不是**C#**关键字。除了这里描述的**#region**和**#endregion**关键字外，其他关键字都相当复杂，也都有专门的用途。所以在阅读完本书后，读者可以再去研究这个主题。

现在不必考虑示例中的其他代码，因为本书前几章仅解释**C#**的基本语法，至于应用程序进行**Console.WriteLine()**调用的具体方式，则不在我们的考虑之列。以后会阐述这行代码的重要性。

3.3 变量

如前所述，变量关系到数据的存储。实际上，可以把计算机内存中的变量看成架子上的盒子。在这些盒子中，可以放入一些东西，再把它们取出来，或者只是看看盒子里是否有东西。变量也是这样，数据可放在变量中，可以从变量中取出数据或查看它们。

尽管计算机中的所有数据事实上都是相同的东西（一组0和1），但变量有不同的内涵，称为类型。下面再用盒子来类比，盒子有不同的形状和尺寸，某些东西只适合放在特定的盒子中。建立这个类型系统的原因是，不同类型的数据需要用不同的方法来处理。将变量限定为不同的类型可以避免混淆。例如，组成数字图片的0和1序列与组成声音文件的0和1序列，其处理方式是不同的。

要使用变量，需要声明它们，即给变量指定名称和类型。声明变量后，就可以把它们用作存储单元，存储所声明的数据类型的数据。

声明变量的C#语法是指定类型和变量名，如下所示：

```
<type
```

```
><name
```

```
>;
```


如果使用未声明的变量，代码将无法编译，但此时编译器会告诉我们出现了什么问题，所以这不是一个灾难性错误。另外，使用未赋值的变量也会产生一个错误，编译器会检测出这个错误。

3.3.1 简单类型

简单类型就是组成应用程序中基本构件的类型，例如，数值和布尔值（true或false）。与复杂类型不同，简单类型没有子类型或特性。大多数简单类型都是存储数值的，初看起来有点奇怪，使用一种类型来存储数值不可以吗？

有很多数值类型是因为在计算机内存中，把数字作为一系列的0和1来存储。对于整数值，用一定的位（单个数字，可以是0或1）来存储，用二进制格式来表示。以N位来存储的变量可以表示任何介于0到（ $2^N - 1$ ）之间的数。大于这个值的数因为太大，所以无法存储在这个变量中。

例如，有一个变量存储了两位，在整数和表示该整数的位之间的映射应如下所示：

0 = 00

1 = 01

2 = 10

3 = 11

如果要存储更多数字，就需要更多的位（例如，3位可以存储0到7的数）。

这样得到的结论是要存储每个可以想象得到的数，就需要非常多的位，这并不适合PC。即使可以用足够多的位来表示每一个数，用这么多的位存储一个表示范围很小的变量（例如0到10）的效率非常低下，因为存储器被浪费了。其实表示0到10之间的数，4位就足够了，这样就可以用相同的内存空间存储这个范围内的更多数值。

相反，许多不同的整数类型可用于存储不同范围的数值，占用不同的内存空间（至多64位），这些类型如表3-1所示。

表3-1 整数类型

类型	别名	允许的值
sbyte	System.SByte	介于-128和127之间的整数
byte	System.Byte	介于0和255之间的整数
short	System.Int16	介于-32 768和32 767之间的整数
ushort	System.UInt16	介于0和65 535之间的整数
int	System.Int32	介于-2 147 483 648和2 147 483 647之间的整数
uint	System.UInt32	介于0和4 294 967 295之间的整数
long	System.Int64	介于-9 223 372 036 854 775 808和9 223 372 036 854 775 807之间的整数
ulong	System.UInt64	介于0和18 446 744 073 709 551 615之间的整数

注意： 这些类型中的每一种都利用了.NET Framework中定义的标准类型。如第1章所述，使用标准类型可以在语言之间交互操作。在C#中这些类型的名称是Framework中定义的地类型的别名，表3-1列出了这些类型在.NET Framework库中的名称。

一些变量名称前面的“u”是unsigned的缩写，表示不能在这些类型的变量中存储负数，参见表3-1中的“允许的值”一列。

当然，还需要存储浮点数，它们不是整数。可以使用的浮点数变量类型有3种：float、double和decimal。前两种可以用 $\pm m \times 2^e$ 的形式存储浮点数，m和e的值因类型而异。decimal使用另一种形式： $\pm m \times 10^e$ 。这3种类型、m和e的值，以及它们在实数中的上下限如表3-2所示。

表3-2 浮点类型

类 型	别 名	m 的 最小值	m 的 最大值	e 的 最小值	e 的 最大值	近似的 最小值	近似的 最大值
float	System.Single	0	2^{24}	-149	104	1.5×10^{-45}	3.4×10^{38}
double	System.Double	0	2^{53}	-1075	970	5.0×10^{-324}	1.7×10^{308}
decimal	System.Decimal	0	2^{96}	-28	0	1.0×10^{-28}	7.9×10^{28}

除数值类型外，另外还有3种简单类型，如表3-3所示。

表3-3 文本和布尔类型

类型	别名	允许的值
		一个Unicode字符，存储0和65 535之间的整

char	System.Char	数
bool	System.Boolean	布尔值：true或false
string	System.String	一组字符

注意组成string的字符数量没有上限，因为它可以使用可变大小的内存。

布尔类型bool是C#中最常用的一种变量类型，类似的类型在其他语言的代码中非常丰富。当编写应用程序的逻辑流程时，一个可以是true或false的变量有非常重要的分支作用。例如，考虑一下有多少问题可以用true或false（或yes和no）来回答。执行变量值之间的比较或检查输入的有效性就是后面使用布尔变量的两个编程示例。

介绍了这些类型后，下面用一个简短示例来声明和使用它们。在下面的示例中，要使用一些简单的代码来声明两个变量，给它们赋值，再输出这些值。

试一试：使用简单类型的变量：**Ch03Ex01\Program.cs**

（1）在目录C:\BegVCSharp\Chapter03下创建一个新的控制台应用程序Ch03Ex01。

（2）在Program.cs中添加如下代码：

```
static void Main(string[] args)
```

```
{  
    int myInteger;  
  
    string myString;  
  
    myInteger = 17;  
  
    myString = "\"myInteger\" is";  
  
    Console.WriteLine($"{myString} {myInteger}");  
  
    Console.ReadKey();  
  
}
```

(3) 运行代码，结果如图3-1所示。

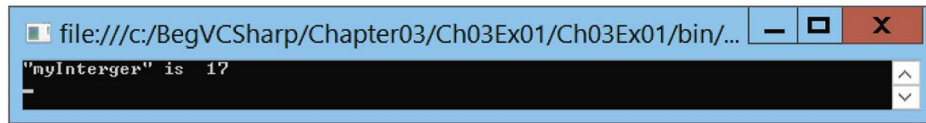


图3-1

示例说明

我们添加的代码完成了以下3项任务：

- 声明两个变量
- 给这两个变量赋值
- 将这两个变量的值输出到控制台

变量声明使用下述代码：

```
int myInteger;  
string myString;
```

第一行声明一个类型为int的变量myInteger，第二行声明一个类型为string的变量myString。

注意： 变量的命名是有限制的，不能使用任意字符序列。“变量的命名”一节将介绍变量的命名规则。

接下来的两行代码为变量赋值：

```
myInteger = 17;  
myString = "\"myInteger\" is";
```

使用=赋值运算符（在本章的“表达式”一节中将详细介绍）给变量分配两个固定的值（在代码中称为字面值）。把整数值17赋给myInteger，把字符串"myInteger\" is（包括引号）赋给myString。

```
"myInteger" is
```

以这种方式给字符串赋予字面值时，必须用双引号把字符串括起来。因此，如果字符串本身包含双引号，就会出现错误，必须用一些表示这些字符的其他字符（即转义序列）来替代它们。本例使用序列\"来转义双引号：

```
myString = "\"myInteger\" is";
```

如果不使用这些转义序列，而输入如下代码：

```
myString = ""myInteger" is";
```

就会出现编译错误。

注意给字符串赋予字面值时，必须小心换行——C#编译器会拒绝分布在多行上的字符串字面值。要添加一个换行符，可在字符串中使用换行符的转义序列，即\n。例如，赋值语句：

```
myString = "This string has a\nline break.";
```

会在控制台视图中显示两行代码，如下所示：

```
This string has a
```

`line break.`

所有转义序列都包含一个反斜杠符号，后跟一个字符组合（详见后面的内容），因为反斜杠符号的这种用途，它本身也有一个转义序列，即两个连续的反斜杠\\。

下面继续解释代码，还有一行没有说明：

```
Console.WriteLine($"{myString} {myInteger}");
```

这是C# 6中的一个新功能，称为字符串插入，它看起来类似于第一个示例中把文本写到控制台的简单方法，但本例指定了变量。这里不详细讨论这行代码，只需要知道这是本书第I部分用于给控制台窗口输出文本的一种技巧。

在后面的示例中，就使用这种给控制台输出文本的方式显示代码的输出结果。最后一行代码在前面的示例中也出现过，用于在程序结束前等待用户输入内容：

```
Console.ReadKey();
```

这里不详细探讨这行代码，但后面的示例会常常用到它。现在只需要知道，它暂停代码的执行，等待用户按下一个键。

3.3.2 变量的命名

如上一节所述，不能把任意序列的字符作为变量名。但没必要为此感到担心，因为这种命名系统仍是非常灵活的。

基本的变量命名规则如下：

- 变量名的第一个字符必须是字母、下划线（_）或@。
- 其后的字符可以是字母、下划线或数字。

另外，有一些关键字对于C#编译器而言具有特定的含义，例如前面出现的using和namespace关键字。如果错误地使用其中一个关键字，编译器会产生一个错误，我们马上就会知道出错了，所以不必担心。

例如，下面的变量名是正确的：

myBigVar

VAR1

_test

下列变量名有误：

99BottlesOfBeer

namespace

It's-All-Over

3.3.3 字面值

在前面的示例中，有两个字面值示例：整数（17）和字符串（`"myInteger" is`）。其他变量类型也有相关的字面值，如表3-4所示。其中有许多涉及后缀，即在字面值的后面添加一些字符来指定想要的类型。一些字面值有多种类型，在编译时由编译器根据它们的上下文确定其类型（同样见表3-4）。

表3-4 字面值

类型	类别	后缀	示例/允许的值
bool	布尔	无	true或false
int、uint、 long、ulong	整数	无	100
uint、ulong	整数	u或U	100U
long、ulong	整数	l或L	100L
ulong	整数	ul、uL、Ul、UL、 lu、lU、Lu或LU	100UL
float	实数	f或F	1.5F
double	实数	无、d或D	1.5
decimal	实数	m或M	1.5M
char	字符	无	'a'或转义序列
string	字符串	无	"a...a"，可以包含 转义序列

字符串字面值

本章前面介绍了几个可在字符串字面值中使用的转义序列，表3-5是这些转义序列的完整列表，以便以后引用。

表3-5 字符串字面值的转义序列

--	--	--

转义序列	产生的字符	字符的Unicode值
\'	单引号	0x0027
\"	双引号	0x0022
\\	反斜杠	0x005C
\0	null	0x0000
\a	警告（产生蜂鸣）	0x0007
\b	退格	0x0008
\f	换页	0x000C
\n	换行	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B

表3-5中的“字符的Unicode值”列是字符在Unicode字符集中的十六进制值。除了上面这些，还可以使用Unicode转义序列指定其他任何Unicode字符，该转义序列包括标准的\字符，后跟一个u和一个4位十六进制值（例如，表3-5中x后面的4位数字）。

下面的字符串是等价的：

```
"Benjamin\'s string."
```

```
"Benjamin\u0027s string."
```

显然，Unicode转义序列还有更多用途。

也可以一字不变地指定字符串，即两个双引号之间的所有字符都包含在字符串中，包括行末字符和原本需要转义的字符。唯一的例外是必须指定双引号字符的转义序列，以免结束字符串。这种方法需要在字符串之前加一个@字符：

```
@ "Verbatim string literal."
```

也可以用普通方式指定这个字符串，但下面的字符串就必须使用@字符：

```
@ "A short list:  
item 1  
item 2"
```

一字不变的字符串在文件名中非常有用，因为文件名中大量使用了反斜杠字符。如果使用一般字符串，就必须在字符串中使用两个反斜杠，例如：

```
"C:\\Temp\\MyDir\\MyFile.doc"
```

而有了一字不变的字符串字面值，这段代码就更便于阅读。下面的字符串与上面的等价：

```
@ "C:\Temp\MyDir\MyFile.doc"
```

注意：从本书的后面可以看出，字符串是引用类型，而本章中的其他类型都是值类型。所以，字符串也可以被赋予null值，表示字符

串变量不引用字符串（或其他任何东西）。

3.4 表达式

C#包含许多执行这类处理的运算符。把变量和字面值（在使用运算符时，它们都称为操作数）与运算符组合起来，就可以创建表达式，它是计算的基本构件。

运算符范围广泛，有简单的，也有非常复杂的，其中一些可能只在数学应用程序中使用。简单的操作包括所有的基本数学操作，例如+运算符是把两个操作数加在一起，而复杂的操作包括通过变量内容的二进制表示来处理它们。还有专门用于处理布尔值的逻辑运算符，以及赋值运算符，如=运算符。

本章主要介绍数学和赋值运算符，而逻辑运算符将在第4章中介绍，因为第4章主要论述控制程序流程的布尔逻辑。

运算符大致分为如下3类：

- 一元运算符，处理一个操作数
- 二元运算符，处理两个操作数
- 三元运算符，处理三个操作数

大多数运算符都是二元运算符，只有几个一元运算符和一个三元运算符，即条件运算符（条件运算符是一个逻辑运算符，详见第4章）。下面首先介绍数学运算符，它包括一元运算符和二元运算符。

3.4.1 数学运算符

有5个简单的数学运算符，其中两个（+和-）有二元和一元两种形式。表3-6列出了这些运算符，并用一个简短示例来说明它们的用法，以及使用简单的数值类型（整数和浮点数）时它们的结果。

表3-6 简单的数学运算符

运算符	类别	示例表达式	结果
+	二元	var1=var2+var3;	var1的值是var2与var3的和
-	二元	var1=var2-var3;	var1的值是从var2减去var3所得的值
*	二元	var1=var2* var3;	var1的值是var2与var3的乘积
/	二元	var1=var2/var3;	var1是var2除以var3所得的值
%	二元	var1=var2 % var3;	var1是var2除以var3所得的余数
+	一元	var1=+var2;	var1的值等于var2的值
-	一元	var1=-var2;	var1的值等于var2的值乘以-1

+（一元）运算符有点古怪，因为它对结果没有影响。它不会把值变成正的：如果var2是-1，则+var2仍是-1。但这是一个得到普遍认可的运算符，所以也把它包含进来。这个运算符最有用的方面是，可

以定制它的操作，本书在后面探讨运算符的重载时会介绍它。

上面的示例都使用简单的数值类型，因为使用其他简单类型，结果可能不太清晰。例如把两个布尔值加在一起，会得到什么结果？因此，如果对bool变量使用+（或其他数学运算符），编译器会报错。char变量的相加也会有点让人摸不着头脑。记住，char变量实际上存储的是数字，所以把两个char变量加在一起也会得到一个数字（其类型为int）。这是一个隐式转换示例，稍后将详细介绍这个主题和显式转换，因为它也可以应用到var1、var2和var3是混合类型的情况。

二元运算符+在用于字符串类型变量时也是有意义的。此时，它的作用如表3-7所示。

表3-7 字符串连接运算符

运算符	类别	示例表达式	结果
+	二元	var1=var2+var3;	var1的值是存储在var2和var3中的两个字符串的连接值

但其他数学运算符不能用于处理字符串。

这里应介绍的另两个运算符是递增和递减运算符，它们都是一元运算符，可通过两种方式加以使用：放在操作数的前面或后面。简单表达式的结果如表3-8所示。

表3-8 简单表达式的结果

--	--	--	--

运算符	类别	示例表达式	结果
++	一元	var1=++var2;	var1的值是var2+1， var2递增1
—	一元	var1=--var2;	var1的值是var2- 1， var2递减1
++	一元	var1=var2++;	var1的值是var2， var2递增1
—	一元	var1=var2--;	var1的值是var2， var2递减1

这些运算符改变存储在操作数中的值。

- ++总是使操作数加1
- —总是使操作数减1

var1中存储的结果有区别，其原因是运算符的位置决定了它什么时候发挥作用。把运算符放在操作数的前面，则操作数是在进行任何其他计算前受到运算符的影响；而如果把运算符放在操作数的后面，则操作数是在完成表达式的计算后受到运算符的影响。

再看一个示例。考虑以下代码：

```
int var1, var2 = 5, var3 = 6;
var1 = var2++ * --var3;
```

要把什么值赋予var1？在计算表达式前，var3前面的运算符—会起作用，把它的值从6改为5。可以忽略var2后面的++运算符，因为它是在计算完成后才发挥作用，所以var1的结果是5与5的乘积，即25。

许多情况下，这些简单的一元运算符使用起来非常方便，它们实际

上是下述表达式的简写形式：

```
var1 = var1 + 1;
```

这类表达式有许多用途，特别适于在循环中使用，这将在第4章讲述。下面的示例说明如何使用数学运算符，并介绍另外两个有用的概念。代码提示用户键入一个字符串和两个数字，然后显示计算结果。

试一试：用数学运算符处理变量：**Ch03Ex02\Program.cs**

(1) 在目录C:\BegVCSharp\Chapter03下创建一个新的控制台应用程序Ch03Ex02。

(2) 在Program.cs中添加如下代码：

```
static void Main(string[] args)
{
    double firstNumber, secondNumber;

    string userName;

    Console.WriteLine("Enter your name:");
```

```
userName = Console.ReadLine();
```

```
Console.WriteLine($"Welcome {userName}!");
```

```
Console.WriteLine("Now give me a number:");
```

```
firstNumber = Convert.ToDouble(Console.ReadLine());
```

```
Console.WriteLine("Now give me another number:");
```

```
secondNumber = Convert.ToDouble(Console.ReadLine());
```

```
Console.WriteLine($"The sum of {firstNumber} and {secondNumber} is {firstNumber + secondNumber}");
```

```
    $"{firstNumber + secondNumber}.");
```

```
Console.WriteLine($"The result of subtracting {secondNumber} from {firstNumber} is {firstNumber - secondNumber}");
```

```
    $"{firstNumber} is {firstNumber - secondNumber}");
```

```
Console.WriteLine($"The product of {firstNumber} and {secondNumber} is {firstNumber * secondNumber}");
```

```
    $"{firstNumber * secondNumber}.");
```

```
Console.WriteLine($"The result of dividing {firstNumber} by {secondNumber} is {firstNumber / secondNumber}");
```

```
${secondNumber} is {firstNumber / secondNumber}
```

```
Console.WriteLine($"The remainder after dividing {firstNumber}
```

```
${secondNumber} is {firstNumber % secondNumber}
```

```
Console.ReadKey();
```

```
}
```

(3) 执行代码，结果如图3-2所示。

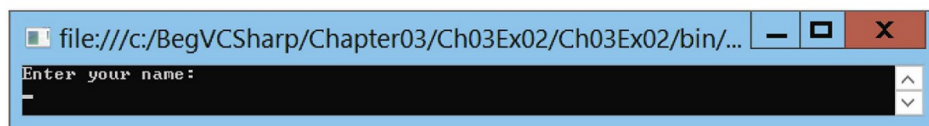


图3-2

(4) 输入名称，按下回车键，如图3-3所示。

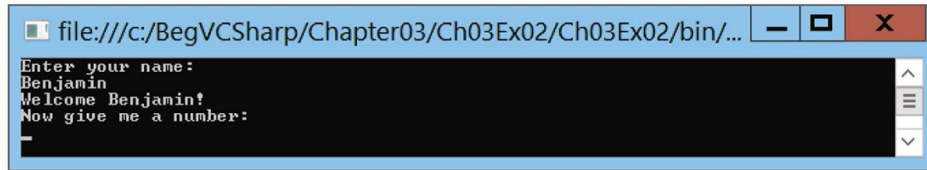


图3-3

(5) 输入一个数字，按下回车键，再输入另一个数字，按下回车键，结果如图3-4所示。

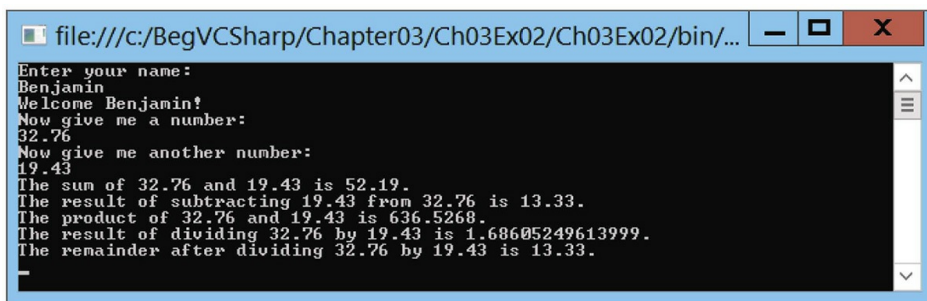


图3-4

示例说明

除了演示数学运算符外，这段代码还引入了两个重要概念，在以后的示例中将多次用到这些概念：

- 用户输入
- 类型转换

用户输入使用与前面`Console.WriteLine()`命令类似的语法。但这里使用`Console.ReadLine()`。这个命令提示用户输入信息，并把它们存储在`string`变量中：

```
string userName;
```

```
Console.WriteLine("Enter your name:");  
userName = Console.ReadLine();  
Console.WriteLine($"Welcome {userName}!");
```

这段代码直接将已赋值变量userName的内容写到屏幕上。

这个示例还读取了两个数字。这有些复杂，因为Console.ReadLine()命令生成一个字符串，而我们希望得到一个数字，所以这就引入了类型转换的问题。第5章将详细讨论类型转换，下面首先分析本例使用的代码。

首先声明要存储数字输入的变量：

```
double firstNumber, secondNumber;
```

接着给出提示，对Console.ReadLine()得到的字符串使用命令Convert.ToDouble()，把字符串转换为double类型，把这个数值赋给前面声明的变量firstNumber：

```
Console.WriteLine("Now give me a number:");  
firstNumber = Convert.ToDouble(Console.ReadLine());
```

这个语法相当简单，其他许多转换也用类似的方式进行。

其余代码按同样方式获取第二个数：

```
Console.WriteLine("Now give me another number:");  
secondNumber = Convert.ToDouble(Console.ReadLine());
```

然后输出两个数字加、减、乘、除的结果，并用余数运算符（%）

显示除操作的余数：

```
Console.WriteLine($"The sum of {firstNumber} and {secondNumber}
    $"{firstNumber + secondNumber}.");
Console.WriteLine($"The result of subtracting {secondNumber} f
    $"{firstNumber} is {firstNumber - secondNumber}.");
Console.WriteLine($"The product of {firstNumber} and {secondNu
    $"is {firstNumber * secondNumber}.");
Console.WriteLine($"The result of dividing {firstNumber} by "
    $"{secondNumber} is {firstNumber / secondNumber}.");
Console.WriteLine($"The remainder after dividing {firstNumber}
    $"{secondNumber} is {firstNumber % secondNumber}.");
```

注意，我们提供了表达式`firstNumber+secondNumber`等，作为`Console.WriteLine()`语句的一个参数，而没有使用中间变量：

```
Console.WriteLine($"The sum of {firstNumber} and {secondNumber}
    $"{firstNumber + secondNumber}.");
```

这种语法可以提高代码的可读性，并减少需要编写的代码量。

3.4.2 赋值运算符

我们迄今一直在使用简单的`=`赋值运算符，其实还有其他赋值运算符，而且它们都很有用。除了`=`运算符外，其他赋值运算符都以类似方式工作。与`=`一样，它们都是根据运算符和右边的操作数，把一个值赋给左边的变量。

表3-9列出了这些运算符及其说明。

表3-9 赋值运算符

运算符	类别	示例表达式	结果
=	二元	var1=var2;	var1被赋予var2的值
+=	二元	var1+=var2;	var1被赋予var1与var2的和
-=	二元	var1-=var2;	var1被赋予var1与var2的差
=	二元	var1=var2;	var1被赋予var1与var2的乘积
/=	二元	var1/=var2;	var1被赋予var1与var2相除所得的结果
%=	二元	var1 %=var2;	var1被赋予var1与var2相除所得的余数

可以看出，这些运算符把var1也包括在计算过程中，下面的代码：

```
var1 += var2;
```

与下面的代码结果相同。

```
var1 = var1 + var2;
```

注意： 与+运算符一样，+=运算符也可以用于字符串。

使用这些运算符，特别是在使用长变量名时，可使代码更便于阅读。

3.4.3 运算符的优先级

在计算表达式时，会按顺序处理每个运算符。但这并不意味着必须从左至右地运用这些运算符。例如，考虑下面的代码：

```
var1 = var2 + var3;
```

其中+运算符就是在=运算符之前进行计算的。在其他一些情况下，运算符的优先级并没有这么明显，例如：

```
var1 = var2 + var3 * var4;
```

其中*运算符首先计算，其后是+运算符，最后是=运算符，这是标准的数学运算顺序，其结果与我们在纸上进行算术运算的结果相同。

像这样的计算，可以使用括号控制运算符的优先级，例如：

```
var1 = (var2 + var3) * var4;
```

首先计算括号中的内容，即+运算符在*运算符之前计算。

对于前面介绍的运算符，其优先级如表3-10所示，优先级相同的运算符（如*和/）按照从左至右的顺序计算。

表3-10 运算符的优先级

优 先 级	运 算 符
优先级由高到低	++, --(用作前缀)、+、- (一元)
	*, /、%
	+、-
	=、*=、/=、%=、+=、-=
	++, --(用作后缀)

注意： 如上所述，括号可用于重写优先级顺序。另外，++和—用作后缀运算符时，在概念上其优先级最低，如上表所示。它们不对赋值表达式的结果产生影响，所以可以认为它们的优先级比所有其他运算符都高。但是，它们会在计算表达式后改变操作数的值，所以认为它们的优先级如表3-10所示会十分方便。

3.4.4 名称空间

在继续学习前，应花一定的时间了解一个比较重要的主题——名称空间。它们是.NET中提供应用程序代码容器的方式，这样就可以唯一地标识代码及其内容。名称空间也用作.NET Framework中给项分类的一种方式。大多数项都是类型定义，例如，本章描述的简单类型（`System.Int32`等）。

默认情况下，C#代码包含在全局名称空间中。这意味着对于包含在这段代码中的项，全局名称空间中的其他代码只要通过名称进行引用，就可以访问它们。可使用`namespace`关键字为花括号中的代码块显式定义名称空间。如果在该名称空间代码的外部使用名称空间中的名称，就

必须写出该名称空间中的限定名称。

限定名称包括它所有的分层信息。这意味着，如果一个名称空间中的代码需要使用在另一个名称空间中定义的名称，就必须包括对该名称空间的引用。限定名称在不同的名称空间级别之间使用句点字符（.），如下所示：

```
namespace LevelOne
{
    // code in LevelOne namespace
    // name "NameOne" defined
}
// code in global namespace
```

这段代码定义了一个名称空间LevelOne，以及该名称空间中的一个名称NameOne（注意这里在应该定义名称空间的地方添加了一个注释，而没有列出实际代码，这是为了使我们的讨论更具普遍性）。在名称空间LevelOne中编写的代码可以直接使用NameOne来引用该名称，但全局名称空间中的代码必须使用限定名称LevelOne.NameOne来引用这个名称。

需要注意特别重要的一点：**using**语句本身不能访问另一个名称空间中的名称。除非名称空间中的代码以某种方式链接到项目上，或者代码是在该项目的源文件中定义的，或者是在链接到该项目的其他代码中定义的，否则就不能访问其中包含的名称。另外，如果包含名称空间的代码链接到项目上，那么无论是否使用**using**，都可以访问其中包含的名称。**using**语句便于我们访问这些名称，减少代码量，以及提高可读性。

回头分析本章开头的ConsoleApplication1中的代码，会看到下面这些被应用到名称空间上的代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
    ...
}
```

以using关键字开头的5行代码声明在这段C#代码中使用System、System.Collections.Generic、System.Linq、System.Text和System.Threading.Tasks名称空间，它们可以在该文件的所有名称空间中访问，不必进行限定。System名称空间是.NET Framework应用程序的根名称空间，包含控制台应用程序需要的所有基本功能。其他4个名称空间常用于控制台应用程序，所以该程序包含了它们。最后，为应用程序代码本身声明一个名称空间ConsoleApplication1。

C# 6新增了using static关键字。这个关键字允许把静态成员直接包含到C#程序的作用域中。例如，本章的两个示例都使用了System.Console静态类中的System.Console.WriteLine()方法。注意，在这些例子中，应包括Console类和WriteLine()方法。把using static System.Console添加到名称空间列表中时，访问WriteLine()方法就不再需要在前面加上静态类名。

之后需要System.Console静态类的代码示例就包括using static System.Console关键字。

3.5 练习

(1) 在下面的代码中，如何从名称空间fabulous的代码中引用名称great?

```
namespace fabulous
{
    // code in fabulous namespace
}
namespace super
{
    namespace smashing
    {
        // great name defined
    }
}
```

(2) 下面哪些变量名不合法?

- myVariableIsGood
- 99Flake
- _floor
- time2GetJiggyWidIt
- wrox.com

(3) 字符串"supercalifragilisticexpialidocious"是不是太长了，不能

放在string变量中？如果是，原因是什么？

（4）考虑运算符的优先级，列出下述表达式的计算步骤：

```
resultVar += var1 * var2 + var3 % var4 / var5;
```

（5）编写一个控制台应用程序，要求用户输入4个int值，并显示它们的乘积。提示：前面看到可以使用Convert.ToDouble()命令把用户在控制台上输入的数转换为double类型；类似地，从string类型转换为int类型的命令是Convert.ToInt32()。

附录A给出了练习答案。

3.6 本章要点

主题	要点
C#基本语法	C#是一种区分大小写的语言，每行代码都以分号结束。如果代码行太长或者想要标识嵌套的块，可以缩进代码行，以方便阅读。使用//或/*...*/语法可以包含不编译的注释。代码块可以隐藏到区域中，也是为了方便阅读
变量	变量是有名称和类型的数据块。.NET Framework定义了大量简单类型，例如数字和字符串（文本）类型，以供使用。变量只有经过声明和初始化后，才能使用。可以把字面值赋予变量，以初始化它们，变量还可在单个步骤中声明和初始化
表达式	表达式利用运算符和操作数来建立，其中运算符对操作数执行操作。运算符有3种：一元、二元和三元运算符，它们分别操作1、2和3个操作数。数学运算符对数值执行操作，赋值运算符把表达式的结果放在变量中。运算符有固定的优先级，优先级确定了运算符在表达式中的处理顺序
名称空间	.NET应用程序中定义的所有名称，包括变量名，都包含在名称空间中。名称空间采用层次结构，我们通常需要根据包含名称的名称空间来限定名称，以便访问它们

第4章 流程控制

本章内容：

- 布尔逻辑的用法
- 如何控制代码的分支
- 如何编写循环代码

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 4 Code后，可以找到与本章示例对应的单独文件。

我们迄今看到的C#代码有一个共同点：程序的执行都是一行接一行、自上而下地进行，不遗漏任何代码。如果所有应用程序都这样执行，我们能做的工作就很有限了。本章介绍控制程序流程的两种方法。程序流程就是C#代码的执行顺序。这两种方法是分支和循环。分支根据计算的结果有条件地执行代码，例如，“只有myVal小于10，才执行这行代码”。循环重复执行相同的语句（重复执行一定的次数，或在满足测试条件后才停止执行）。

这两种方法都用到布尔逻辑。第3章介绍了bool类型，但并未讨论

它。本章将在很多地方使用它，所以先讨论布尔逻辑，以便在流程控制环境下使用它。

4.1 布尔逻辑

第3章介绍的bool类型可以有两个值：true或false。这种类型常用于记录某些操作的结果，以便操作这些结果。特别是，bool类型可用于存储比较的结果。

注意： 19世纪中叶的英国数学家乔治·布尔（George Boole）为布尔逻辑奠定了基础。

考虑下述情形（如本章引言所述）：要根据变量myVal是否小于10来确定是否执行代码。为此，需要确定语句“myVal小于10”的真假，即需要了解比较的布尔结果。

布尔比较需要使用布尔比较运算符（也称为关系运算符），如表4-1所示。

表4-1 布尔比较运算符

运算符	类别	示例表达式	结果
==	二元	var1=var2==var3;	如果var2等于var3，var1的值就是true，否则为false
!=	二元	var1=var2 !=var3;	如果var2不等于var3，var1的值就是true，否则为false

<	二元	<code>var1=var2<var3;</code>	如果var2小于var3，var1的值就是true，否则为false
>	二元	<code>var1=var2>var3;</code>	如果var2大于var3，var1的值就是true，否则为false
<=	二元	<code>var1=var2<=var3;</code>	如果var2小于等于var3，var1的值就是true，否则为false
>=	二元	<code>var1=var2>=var3;</code>	如果var2大于等于var3，var1的值就是true，否则为false

在表4-1中，var1都是bool类型的变量，var2和var3则可以是不同类型。

在代码中，可以对数值使用这些运算符：

```
bool isLessThan10;
isLessThan10 = myVal < 10;
```

如果myVal存储的值小于10，这段代码就给isLessThan10赋予true值，否则赋予false值。

也可以对其他类型使用这些比较运算符，例如字符串：

```
bool isBenjamin;
isBenjamin = myString == "Benjamin";
```

如果myString存储的字符串是Benjamin，isBenjamin的值就为true。

也可以对布尔值使用这些运算符：

```
bool isTrue;
isTrue = myBool == true;
```

但只能使用==和!=运算符。

注意： 一个常见的代码错误是，无意间假定由于val1<val2是false，所以val1>val2为true。如果val1==val2，则这两条语句都是false。

&和|运算符也有两个类似的运算符，称为条件布尔运算符（见表4-2）。

表4-2 条件布尔运算符

运算符	类别	示例表达式	结果
&&	二元	var1=var2 && var3;	如果var2和var3都是true，var1的值就是true，否则为false（逻辑与）
	二元	var1=var2 var3;	如果var2或var3是true（或两者都是），var1的值就是true，否则为false（逻辑或）

这些运算符的结果与&和|完全相同，但得到结果的方式有一个重要区别：其性能比较好。两者都是检查第一个操作数的值（表4-2中的var2），如果已经能判断结果，就根本不处理第二个操作数（表4-2中的var3）。

如果&&运算符的第一个操作数是false，就不需要考虑第二个操作数的值了，因为无论第二个操作数的值是什么，其结果都是false。同样，如果第一个操作数是true，||运算符就返回true，不必考虑第二个操作数的值。

4.1.1 布尔按位运算符和赋值运算符

使用布尔赋值运算符可以把布尔比较与赋值组合起来，其方式与第3章中的数学赋值运算符（+=、*=等）相同。布尔赋值运算符如表4-3所示。当表达式使用赋值（=）和按位运算符（&、|、^）时，就使用所比较数值的二进制表示来计算结果，而不是使用整数、字符串或相似的值。

表4-3 布尔赋值运算符

运算符	类别	示例表达式	结果
&=	二元	var1 &=var2;	var1的值是var1 & var2的结果
=	二元	var1 =var2;	var1的值是var1 var2的结果
^=	二元	var1 ^=var2;	var1的值是var1 ^ var2的结果

例如，等式var1 ^=var2类似于var1=var1 ^ var2，其中var1=true、var2=false。当比较false的二进制表示0000与true（一般不是0000的任何值，通常是0001）时，var1就设置为true。

--

注意：&=和|=赋值运算符并不使用&&和||条件布尔运算符，即无论赋值运算符左边的值是什么，都处理所有操作数。

在下面的示例中，用户键入一个整数，然后代码使用该整数执行各种布尔运算。

试一试：使用布尔运算符: **Ch04Ex01\Program.cs**

(1) 在目录C:\BegVCSharp\Chapter04下创建一个新的控制台应用程序Ch04Ex01。

(2) 将以下代码添加到Program.cs中：

```
static void Main(string[] args)
{
    WriteLine("Enter an integer:");

    int myInt =.ToInt32(ReadLine());

    bool isLessThan10 = myInt<10;
```



```
bool isBetween0And5 = (0<= myInt) && (myInt<= 5);
```

```
WriteLine($"Integer less than 10? {isLessThan10}");
```

```
WriteLine($"Integer between 0 and 5? {isBetween0And5}
```

```
WriteLine($"Exactly one of the above is true?
```

```
{isLessThan10 ^ isBetween0And5}");
```

```
ReadKey();
```

```
}
```

(3) 运行应用程序，出现提示时，输入一个整数，结果如图4-1所示。

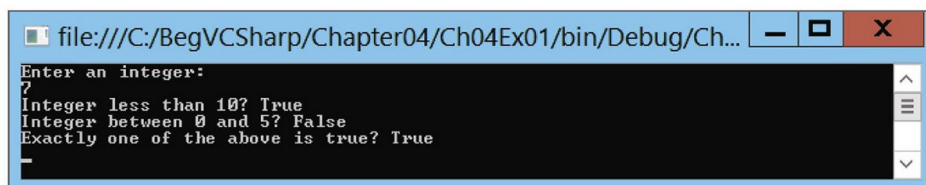


图4-1

示例说明

前两行代码使用前面介绍的技术，提示并接受一个整数值：

```
WriteLine("Enter an integer:");  
int myInt =.ToInt32(ReadLine());
```

使用`Convert.ToInt32()`从字符串输入中得到一个整数。`Convert.ToInt32()`是另一个类型转换命令，与前面使用的`Convert.ToDouble()`命令属于同一系列。`ToInt32()`和`ToDouble()`方法是`System.Convert`静态类的一部分。如第3章所述，自从C# 6之后，就可以在包括的名称空间列表中包含`using static System.Convert`类，直接访问静态类（在这个例子中是`System.Convert`）的方法。还要注意，没有检查用户是否输入了一个整数。如果提供了不是整数的值，例如字符串，在试图执行转换时会发生异常。可以使用`try{ }...catch{ }`块处理这种情况，或在执行转换之前使用`GetType()`方法，检查输入的值是不是一个整数。这两种方法将在后续章节讨论。

接着声明两个布尔变量`isLessThan10`和`isBetween0And5`，并赋值，

其中的逻辑匹配其名称中的描述：

```
bool isLessThan10 = myInt<10;  
bool isBetween0And5 = (0<= myInt) && (myInt<= 5);
```

接着在下面的3行代码中使用这些变量，前两行代码输出它们的值，第3行对它们执行一个操作，并输出结果。在执行这段代码时，假定用户输入了7，如图4-1所示。

第一个输出是操作`myInt<10`的结果。如果`myInt`是6，则它小于10，因此结果为`true`。如果`MyInt`的值是10或更大，就会得到`false`。

第二个输出涉及较多计算：`(0<=myInt) && (myInt<=5)`，其中包含两个比较操作，用于确定`myInt`是否大于或等于0，且小于或等于5。接着对结果进行布尔AND操作。输入数字6，则`(0<=myInt)`返回`true`，而`(myInt<=5)`返回`false`，最终结果就是`(true) && (false)`，即`false`，如图4-1所示。

最后，对两个布尔变量`isLessThan10`和`isBetween0And5`执行逻辑异或操作。如果一个变量的值是`true`，另一个是`false`，则代码返回`true`。所以只有`myInt`是6、7、8或9，才返回`true`，本例输入的是6，所以结果是`true`。

4.1.2 运算符优先级的更新

现在要考虑更多的运算符，所以应更新第3章中的运算符优先级表，把它们包括在内，如表4-4所示。

表4-4 运算符优先级（更新后）

优 先 级	运 算 符
优 先 级 由 高 到 低	++,-- (用作前缀); (), +, - (一元), !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&
	^
	&&
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =
	++, -- (用作后缀)

该表增加了好几个级别，但它明确定义了下述表达式该如何计算：

```
var1 = var2<= 4 && var2 >= 2;
```

其中&&运算符在<=和>=运算符之后执行（在这行代码中，var2是一个int值）。

这里要注意的是，添加括号可以使这样的表达式看起来更清晰。编译器知道用什么顺序执行运算符，但人们常会忘记这个顺序（有时可能想改变这个顺序）。上述表达式也可以写为：

```
var1 = (var2<= 4) && (var2 >= 2);
```

通过明确指定计算的顺序就解决了这个问题。

4.2 分支

分支是控制下一步要执行哪行代码的过程。要跳转到的代码行由某个条件语句来控制。这个条件语句使用布尔逻辑，对测试值和一个或多个可能的值进行比较。

本节介绍C#中的3种分支技术：

- 三元运算符
- if语句
- switch语句

4.2.1 三元运算符

最简单的比较方式是使用第3章介绍的三元（或条件）运算符。一元运算符有一个操作数，二元运算符有两个操作数，所以三元运算符有3个操作数。其语法如下：

```
<test
```

```
> ?<resultIfTrue
```

```
>:<resultIfFalse
```

>

其中，计算<test>可得到一个布尔值，运算符的结果根据这个值来确定是<resultIfTrue>还是<resultIfFalse>。

使用三元运算符可以测试int变量myInteger的值：

```
string resultString = (myInteger<10) ? "Less than 10"  
                        : "Greater than or equal to 10";
```

三元运算符的结果是两个字符串中的一个，这两个字符串都可能赋给resultString。把哪个字符串赋给resultString，取决于myInteger的值与10的比较结果。如果myInteger的值小于10，就把第一个字符串赋给resultString；如果myInteger的值大于或等于10，就把第二个字符串赋给resultString。例如，如果myInteger的值是4，则resultString的值就是字符串"Less than 10"。

4.2.2 if语句

if语句的功能比较多，是有效的决策方式。与?:语句不同的是，if语句没有结果（所以不在赋值语句中使用它），使用该语句是为了根据条件执行其他语句。

if语句最简单的语法如下：

```
if (<test
```

>)

<code executed if

<test

> is true

>;

先执行<test>（其计算结果必须是一个布尔值，这样代码才能编译），如果<test>的计算结果是true，就执行该语句之后的代码。这段代码执行完毕后，或者因为<test>的计算结果是false，而没有执行这段代码，将继续执行后面的代码行。

也可将else语句和if语句合并使用，指定其他代码。如果<test>的计算结果是false，就执行else语句：

if (<test

>)

<code executed if<test> is true>;

else

<code executed if

<test

> is false>;

可使用成对的花括号将这两段代码放在多个代码行上:

if (<test

>)

{

<code executed if<test> is true>

;

}

else

{

<code executed if<test> is false


```
>;  
}
```

例如，重新编写上一节使用三元运算符的代码：

```
string resultString = (myInteger<10) ? "Less than 10"  
                        : "Greater than or equal to 10";
```

因为if语句的结果不能赋给一个变量，所以要单独将值赋给变量：

```
string resultString;  
if (myInteger<10)  
    resultString = "Less than 10";  
else  
    resultString = "Greater than or equal to 10";
```

这样的代码尽管比较冗长，但与三元运算符相比，更便于阅读和理解，也更加灵活。

下面的示例演示了if语句的用法。

试一试：使用if语句： **Ch04Ex02\Program.cs**

(1) 在目录C:\BegVCSharp\Chapter04中创建一个新的控制台应用程序Ch04Ex02。

(2) 把下列代码添加到Program.cs中:

```
static void Main(string[] args)
{
    string comparison;

    WriteLine("Enter a number:");

    double var1 = ToDouble(ReadLine());

    WriteLine("Enter another number:");

    double var2 = ToDouble(ReadLine());

    if (var1<var2)
```

```
comparison = "less than";
```

```
else
```

```
{
```

```
if (var1 == var2)
```

```
comparison = "equal to";
```

```
else
```

```
comparison = "greater than";
```

```
}
```

```
WriteLine($"The first number is
```

```
{comparison} the second number.");
```

```
ReadKey();
```

```
}
```

(3) 执行代码，根据提示输入两个数字，如图4-2所示。

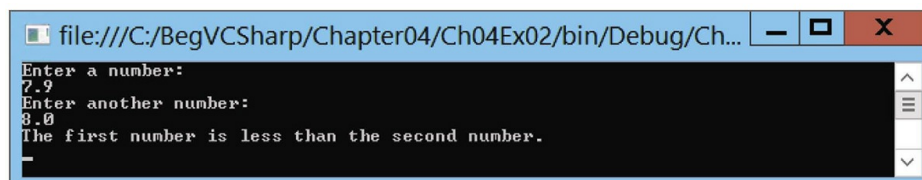


图4-2

示例说明

我们已经十分熟悉代码的第一部分，它从用户输入中得到两个double值：

```
string comparison;  
WriteLine("Enter a number:");  
double var1 = ToDouble(ReadLine());  
WriteLine("Enter another number:");  
double var2 = ToDouble(ReadLine());
```

接着根据var1和var2的值，将一个字符串赋给string变量comparison。首先看看var1是否小于var2：

```
if (var1<var2)  
    comparison = "less than";
```

如果不是，则var1大于或等于var2。在第一个比较操作的else部分，需要嵌套第二个比较：

```
else  
{  
    if (var1 == var2)  
        comparison = "equal to";
```

只有在var1大于var2时，才执行第二个比较操作中的else部分：

```
        else  
            comparison = "greater than";  
    }
```

最后将比较操作的值写到控制台：

```
WriteLine("The first number is {0} the second number.", compar
```

这里使用的嵌套只是进行这些比较的一种方式，还可以编写如下代码：

```
if (var1<var2)
    comparison = "less than";
if (var1 == var2)
    comparison = "equal to";
if (var1 > var2)
    comparison = "greater than";
```

这种方式的缺点在于：无论var1和var2的值是什么，都要执行3个比较操作。在第一种方式中，如果var1<var2是true，就只执行一个比较操作，否则就要执行两个比较操作（还执行了var1==var2比较操作），这样将使执行的代码行较少。在本例中性能上的差异较小，但在较重视速度的应用程序中，性能的差异就很明显了。

使用if语句判断更多条件

在上面的示例中，检查了涉及var1的值的3个条件，包括这个变量所有可能的值。有时要检查特定的值，例如var1是否等于1、2、3或4等。使用上面那样的代码会得到很多烦人的嵌套代码：

```
if (var1 == 1)
{
```

```
    // Do something.
}
else
{
    if (var1 == 2)
    {
        // Do something else.
    }
    else
    {
        if (var1 == 3 || var1 == 4)
        {
            // Do something else.
        }
        else
        {
            // Do something else.
        }
    }
}
```

警告：人们经常会错误地将诸如if (var1==3||var1==4)的条件写为if (var1==3||4)。由于运算符具有优先级，因此首先执行==运算符，接着用||运算符处理布尔和数值操作数，就会出现错误。

这些情况下，就要使用稍有不同的缩进模式，缩短else代码块（即在else块的后面使用一行代码而不是一个代码块），这样就得到了else if语句结构：

```
if (var1 == 1)
{
    // Do something.
}
else if (var1 == 2)
{
    // Do something else.
}
else if (var1 == 3 || var1 == 4)
{
    // Do something else.
}
else
{
    // Do something else.
}
```

这些else if语句实际上是两个独立语句，它们的功能与上述代码相同，但更便于阅读。像这样进行多个比较的操作，应考虑使用另一种分支结构：switch语句。

4.2.3 switch语句

switch语句非常类似于if语句，因为它也是根据测试的值来有条件地执行代码。但是，switch语句可以一次将测试变量与多个值进行比较，而不是仅测试一个条件。这种测试仅限于离散的值，而不是像“大于X”这样的子句，所以它的用法有点不同，但它仍是一种强大的技术。

switch语句的基本结构如下：

```
switch (<testVar

>)
{
    case<comparisonVal1

>:
    <code to execute if

<testVar

> ==<comparisonVal1

> >
    break;
```

case<comparisonVal2

>:

<code to execute if

<testVar

> ==<comparisonVal2

> >

break;

...

case<comparisonValN

>:

<code to execute if

<testVar

> ==<comparisonValN

> >

```
        break;  
default:  
    <code to execute if
```

<testVar

> != comparisonVals

>

```
        break;  
    }
```

<testVar>中的值与每个<comparisonValX>值（在case语句中指定）进行比较，如果有一个匹配，就执行为该匹配提供的语句。如果没有匹配，但有default语句，就执行default部分的代码。

执行完每个部分的代码后，还需要有另一个语句break。在执行完一个case块后，再执行第二个case语句是非法的。

注意： 在此，C#与C++是有区别的。在C++中，可以在运行完一

一个case语句后，运行另一个case语句。

这里的break语句将中断switch语句的执行，而执行该结构后面的语句。

在C#代码中，还有其它方法可以防止程序流程从一个case语句转到下一个case语句。可以使用return语句，中断当前函数的运行，而不是仅中断switch结构的执行（详见第6章）。也可以使用goto语句（如前所述），因为case语句实际上是在C#代码中定义的标签。例如：

```
switch (<testVar

>)
{
    case<comparisonVal1

>:
    <code to execute if

<testVar

> ==<comparisonVal1
```

> >

goto case<comparisonVal2

>;

case<comparisonVal2

>:

<code to execute if

<testVar

> ==<comparisonVal2

> >

break;

...

一个case语句处理完后，不能自由进入下一个case语句，但这条规则有一个例外。如果把多个case语句放在一起（堆叠它们），其后加一个代码块，实际上是一次检查多个条件。如果满足这些条件中的任何一个，就会执行代码，例如：

```
switch (<testVar

>)
{
    case<comparisonVal1>:

    case<comparisonVal2>:

    <code to execute if

<testVar

> ==<comparisonVal1
```

```
> or
```

```
<testVar
```

```
> ==<comparisonVal2
```

```
> >
```

```
    break;
```

```
    ...
```

注意，这些条件也适用于default语句。default语句不一定要放在比较操作列表的最后，还可以把它和case语句放在一起。用break或return添加一个断点，可确保在任何情况下，该结构都有一条有效的执行路径。

在下面的示例中，将使用switch语句，根据用户为测试字符串输入的值，将不同字符串写到控制台。

试一试：使用switch语句： **Ch04Ex03\Program.cs**

(1) 在目录C:\BegVCSharp\Chapter04中创建一个新的控制台应用程序Ch04Ex03。

(2) 把以下代码添加到Program.cs中:

```
static void Main(string[] args)
{
    const string myName = "benjamin";

    const string niceName = "andrea";

    const string sillyName = "ploppy";

    string name;

    WriteLine("What is your name?");

    name = ReadLine();
```



```
switch (name.ToLower())
```

```
{
```

```
    case myName:
```

```
        WriteLine("You have the same name as me!");
```

```
        break;
```

```
    case niceName:
```

```
        WriteLine("My, what a nice name you have!");
```

```
break;
```

```
case sillyName:
```

```
WriteLine("That's a very silly name.");
```

```
break;
```

```
}
```

```
WriteLine($"Hello {name}!");
```

```
ReadKey();
```

}

(3) 执行代码，输入一个姓名，结果如图4-3所示。

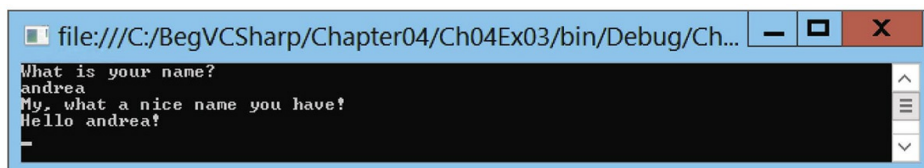


图4-3

示例说明

这段代码建立了3个常量字符串，接受用户输入的一个字符串，再根据输入的字符串把文本写到控制台。这里，字符串是用户输入的姓名。

在比较输入的姓名（在变量name中）和常量值时，首先要用name.ToLower()把输入的姓名转换为小写。name.ToLower()是一个标准命令，可用于处理所有字符串变量，在不能确定用户输入的内容时，使用它是很方便的。使用这个技术，字符串Benjamin、benJamin、benjamin等就会与测试字符串benjamin匹配了。

switch语句尝试将输入的字符串与定义的常量值进行匹配，如果成功，就会用一条个性化的消息问候用户。如果不匹配，则只简单地问候用户。

4.3 循环

循环就是重复执行语句。这个技术使用起来非常方便，因为可以对操作重复任意多次（数千次，甚至数百万次），而不必每次都编写相同的代码。

举一个简单例子，下面的代码计算一个银行账户在10年后的金额，假定支付每年的利息，且该账户没有其他款项的存取：

```
double balance = 1000;
double interestRate = 1.05; // 5% interest/year
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

将相同代码编写10次很费时间，如果把10年改为其他值，又会如何？那就必须把该代码行手工复制需要的次数，这是一件多么痛苦的事！幸运的是，完全不必这样做。使用一个循环就可以对指令执行需要的次数。

循环的另一种重要类型是一直循环到给定的条件满足为止。这些循环比上面描述的循环稍简单些（但同样很有用），所以首先从这类循环开始介绍。

4.3.1 do循环

do循环以下述方式执行：执行标记为循环的代码，然后进行一个布尔测试，如果测试结果为true，就再次执行这段代码，并重复这个过程。当测试结果为false时，就退出循环。

do循环的结构如下：

```
do
{
    <code to be looped>

>
    } while (<Test

>);
```

其中，计算<Test>会得到一个布尔值。

注意： while语句之后必须使用分号。

例如，使用该结构可以把从1到10的数字输出到一列：

```
int i = 1;
do
{
    WriteLine("{0}", i++);
} while (i<= 10);
```

在把i的值写到屏幕上后，使用后缀形式的++运算符递增i的值，所以需要检查一下i<=10，把10也包含在输出到控制台的数字中。

下面的示例使用这个结构略微修改一下本节前面的代码。该段代码计算一个账户在10年后的余额。这次使用一个循环，根据起始的金额和固定利率，计算该账户的金额需要多少年才能达到某个指定的数额。

试一试：使用**do**循环：**Ch04Ex04\Program.cs**

(1) 在目录C:\BegVCSharp\Chapter04创建一个新的控制台应用程序Ch04Ex04。

(2) 把下述代码添加到Program.cs中：

```
static void Main(string[] args)
{
    double balance, interestRate, targetBalance;
```

```
WriteLine("What is your current balance?");
```

```
balance = ToDouble(ReadLine());
```

```
WriteLine("What is your current annual interest rate (in %
```

```
interestRate = 1 + ToDouble(ReadLine()) / 100.0;
```

```
WriteLine("What balance would you like to have?");
```

```
targetBalance = ToDouble(ReadLine());
```

```
int totalYears = 0;
```

```
do
```

```
{
```

```
    balance *= interestRate;
```

```
    ++totalYears;
```

```
}
```

```
while (balance<targetBalance);
```



```
WriteLine($"In {totalYears} year{(totalYears == 1 ? "" : "s
```

```
you'll have a balance of {balance}.");
```

```
ReadKey();
```

```
}
```

(3) 执行代码，输入一些值，示例结果如图4-4所示。

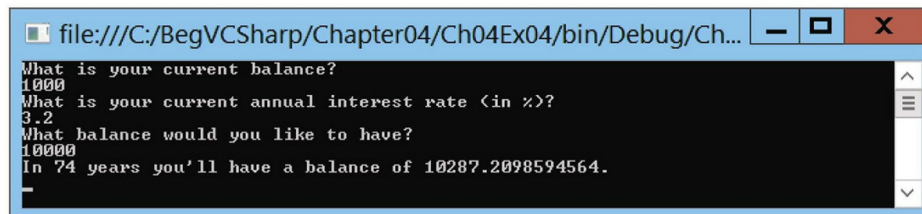


图4-4

示例说明

这段代码利用固定的利率，对年度计算余额的过程重复必要的次数，直到满足结束条件为止。在每次循环中，递增一个计数器变量，就可以确定需要多少年：

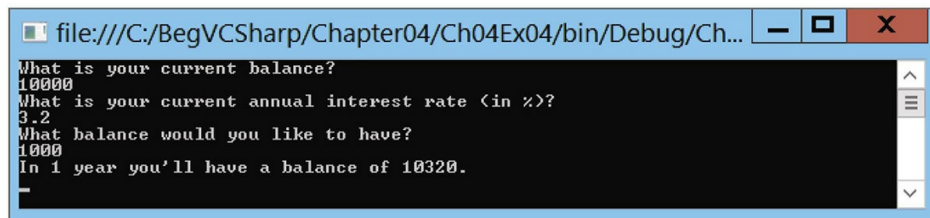
```
int totalYears = 0;
do
{
    balance *= interestRate;
    ++totalYears;
}
while (balance<targetBalance);
```

然后就可以将这个计数器变量用作输出结果的一部分：

```
WriteLine($"In {totalYears}
    year{((totalYears == 1 ? "" : "s"))}
    you'll have a balance of {balance}.");
```

注意： 这可能是?:（三元）运算符最常见的用法了——用最少的代码有条件地格式化文本。这里，如果totalYears不等于1，就在year后面输出一个s。

但这段代码并不完美，考虑一下目标余额少于当前余额的情况，则结果应如图4-5所示。



```
file:///C:/BegVCSharp/Chapter04/Ch04Ex04/bin/Debug/Ch...
What is your current balance?
10000
What is your current annual interest rate (in %)?
3.2
What balance would you like to have?
1000
In 1 year you'll have a balance of 10320.
```

图4-5

do循环至少执行一次。有时（如这个示例）这并不是很理想。当然，可以添加一条if语句：

```
int totalYears = 0;
if (balance<targetBalance)

{
    do
    {
        balance *= interestRate;
        ++totalYears;
    }
    while (balance<targetBalance);
}
WriteLine($"In {totalYears} year{(totalYears == 1 ? "" : "s")}
    $"you'll have a balance of {balance}.");
```

这显然无谓地增加了复杂性。更好的解决方案是使用while循环。

4.3.2 while循环

while循环非常类似于do循环，但有一个明显区别：while循环中的

布尔测试在循环开始时进行，而不是最后进行。如果测试结果为false，就不会执行循环。程序会直接跳转到循环之后的代码。

按下述方式指定while循环：

```
while (<Test  
  
>)  
{  
    <code to be looped  
  
>  
}
```

使用方式几乎与do循环完全相同，例如：

```
int i = 1;  
while (i<= 10)  
  
{  
    WriteLine($"{i++}");  
}
```

这段代码的执行结果与前面的do循环相同，它在一列中输出从1到

10的数字。下面使用while循环修改上一个示例。

试一试：使用while循环： **Ch04Ex05\Program.cs**

(1) 在目录C:\BegVCSharp\Chapter04中创建一个新的控制台应用程序Ch04Ex05。

(2) 修改代码，如下所示（使用Ch04Ex04中的代码作为起点，记住删除原来do循环最后的while语句）：

```
static void Main(string[] args)
{
    double balance, interestRate, targetBalance;
    WriteLine("What is your current balance?");
    balance = ToDouble(ReadLine());
    WriteLine("What is your current annual interest rate (in %)");
    interestRate = 1 + ToDouble(ReadLine()) / 100.0;
    WriteLine("What balance would you like to have?");
    targetBalance = ToDouble(ReadLine());
    int totalYears = 0;
    while (balance < targetBalance)

    {
        balance *= interestRate;
```

```

        ++totalYears;
    }
    WriteLine($"In {totalYears} year{(totalYears == 1 ? "" : "s"}
        $"you'll have a balance of {balance}.");
    if (totalYears == 0)

```

```

        WriteLine(

```

```

            "To be honest, you really didn't need to use this cal

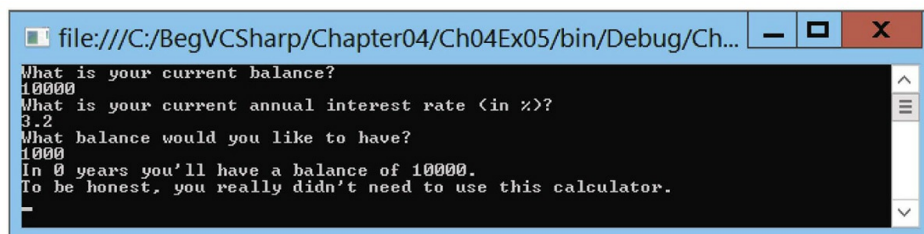
```

```

        ReadKey();
    }

```

(3) 再次执行代码，但这次使用少于起始余额的目标余额，如图4-6所示。



```

file:///C:/BegVCSharp/Chapter04/Ch04Ex05/bin/Debug/Ch...
What is your current balance?
10000
What is your current annual interest rate <in %>?
3.2
What balance would you like to have?
1000
In 0 years you'll have a balance of 10000.
To be honest, you really didn't need to use this calculator.

```

图4-6

示例说明

这段代码只是把do循环改为while循环，就解决了上个示例中的问题。把布尔测试移到开头处，就考虑了不需要执行循环的情况，可以直接跳转到输出结果。

当然，这种情况还有一个解决方案。例如，可以检查用户输入，确保目标余额大于起始余额。此时，可以把用户输入部分放在循环中，如下所示：

```
WriteLine("What balance would you like to have?");
do
{
    targetBalance = ToDouble(ReadLine());
    if (targetBalance<= balance)

        WriteLine("You must enter an amount greater than " +

            "your current balance!\nPlease enter another val

}
}
```

```
while (targetBalance<= balance);
```

这将拒绝接受无意义的值，得到如图4-7所示的结果。

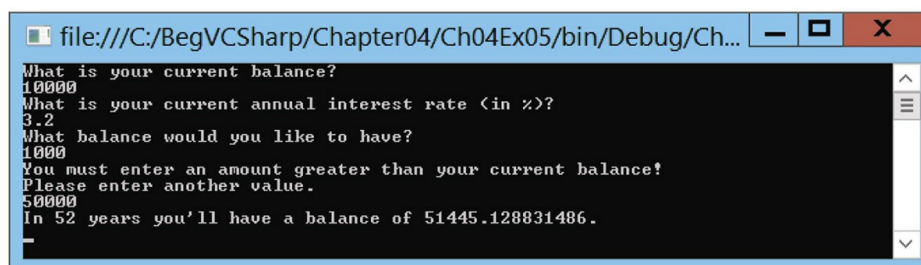


图4-7

在设计应用程序时，用户输入的有效性检查是一个很重要的主题，本书将列举更多这方面的示例。

4.3.3 for循环

本章介绍的最后一类循环是for循环。这类循环可以执行指定的次数，并维护它自己的计数器。要定义for循环，需要下列信息：

- 初始化计数器变量的一个起始值。
- 继续循环的条件，应涉及计数器变量。

- 在每次循环的最后，对计数器变量执行一个操作。

例如，如果要在循环中，使计数器从1递增到10，递增量为1，则起始值为1，条件是计数器小于或等于10，在每次循环的最后，要执行的操作是给计数器加1。

这些信息必须放在for循环的结构中，如下所示：

```
for (<initialization  
  
>;<condition  
  
>;<operation  
  
>)  
    {  
        <code to loop  
  
>  
    }
```

它的工作方式与下述while循环完全相同：

```
<initialization
```

```
>
    while (<condition

>)
    {
        <code to loop

>
        <operation

>
    }
```

前面使用do和while循环输出了从1到10的数字。下面看看如何使用for循环完成这个任务：

```
int i;
for (i = 1; i<= 10; ++i)
{
    WriteLine($"{i}");
}
```

计数器变量是一个整数i，它的初始值是1，在每次循环的最后递增

1. 在每次循环过程中，把i的值写到控制台。

注意，当i的值为11时，将执行循环后面的代码。这是因为在i等于10的循环末尾，i会递增为11。这是在测试条件*i*≤10之前发生的，此时循环结束。与while循环一样，在第一次执行前，只在条件计算为true时才执行for循环，所以可能根本就不会执行循环中的代码。

最后注意，可将计数器变量声明为for语句的一部分，重新编写上述代码，如下所示：

```
for (int i = 1; i<= 10; ++i)
```

```
{  
    WriteLine($"{i}");  
}
```

但如果这么做，就不能在循环外部使用变量i（参见第6章中的“变量的作用域”一节）。

4.3.4 循环的中断

有时需要更精细地控制循环代码的处理。C#为此提供了以下命令：

- **break**——立即终止循环。
- **continue**——立即终止当前的循环（继续执行下一次循环）。
- **return**——跳出循环及包含该循环的函数（参见第6章）。

`break`命令可退出循环，继续执行循环后面的第一行代码，例如：

```
int i = 1;
while (i<= 10)
{
    if (i == 6)
        break;
    WriteLine($"{i++}");
}
```

这段代码输出1-5的数字，因为`break`命令在*i*的值为6时退出循环。

`continue`仅终止当前迭代，而不是整个循环，例如：

```
int i;
for (i = 1; i<= 10; i++)
{
    if ((i % 2) == 0)
        continue;
    WriteLine(i);
}
```

在上面的示例中，只要*i*除以2的余数是0，`continue`语句就终止当前的迭代，所以只显示数字1、3、5、7和9。

4.3.5 无限循环

在代码编写错误或故意进行设计时，可以定义永不终止的循环，即

所谓的无限循环。例如，下面的代码就是无限循环的一个简单例子：

```
while (true)
{
    // code in loop
}
```

有时这种代码也是有用的，而且使用**break**语句或者手工使用Windows任务管理器总是可以退出这样的循环。但当出现这种情形意外时，就会出问题。考虑下面的循环，它与上一节的**for**循环非常类似：

```
int i = 1;
while (i<= 10)
{
    if ((i % 2) == 0)
        continue;
    WriteLine($"{i++}");
}
```

i是在循环的最后一行代码（即**continue**语句后的那条语句）执行完后才递增的。如果程序执行到**continue**语句（此时**i**为2），程序会用相同的**i**值进行下一个循环，然后测试这个**i**值，继续循环，一直这样下去。这就冻结了应用程序。注意仍可以用一般方式退出已冻结的应用程序，所以不必重新启动计算机。

4.4 练习

(1) 如果两个整数存储在变量var1和var2中，该进行什么样的布尔测试，看看其中的一个（但不是两个）是否大于10？

(2) 编写一个应用程序，其中包含练习（1）中的逻辑，要求用户输入两个数字，并显示它们，但拒绝接受两个数字都大于10的情况，并要求用户重新输入。

(3) 下面的代码存在什么错误？

```
int i;
for (i = 1; i<= 10; i++)
{
    if ((i % 2) = 0)
        continue;
    WriteLine(i);
}
```

附录A给出了练习答案。

4.5 本章要点

第5章 变量的更多内容

本章内容：

- 如何在类型之间进行隐式和显式转换
- 如何创建和使用枚举类型
- 如何创建和使用结构类型
- 如何创建和使用数组
- 如何处理字符串值

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 5 Code后，可以找到与本章示例对应的单独文件。

前面介绍了有关C#语言的一些内容，现在将回顾和讨论与变量相关的其他一些较复杂的主题。

首先讨论类型转换，即把值从一种类型转换为另一种类型。前面已经描述了其中的一些信息，这里则要正式讨论。掌握这个主题可以更好地理解表达式中（有意或无意地）混合使用类型时会发生什么，更好地控制处理数据的方式。这有助于理顺代码，避免引起不必要的误解。

接着阐述另一些类型的变量：

- 枚举 ——一种变量类型，用户定义了一组可能的离散值，这些值可以用人们能理解的方式使用。
- 结构 ——一种合成的变量类型，由用户定义的一组其他变量类型组成。
- 数组 ——包含一种类型的多个变量，允许以索引方式访问各个数值。

这些类型比前面使用的简单类型复杂一些，但可以使工作更容易完成。最后，学习另一个与字符串相关的主题——基本字符串处理。

5.1 类型转换

本书前面说过，无论是什么类型，所有数据都是一系列的位，即一系列0和1。变量的含义是通过解释这些数据的方式来确定的。最简单的示例是char类型，这种类型用一个数字表示Unicode字符集中的一个字符。实际上，这个数字与ushort的存储方式完全相同——它们都存储0和65 535之间的数字。

但一般情况下，不同类型的变量使用不同的模式来表示数据。这意味着，即使可以把一系列的位从一种类型的变量移动到另一种类型的变量中（也许它们占用的存储空间相同，也许目标类型有足够的存储空间包含所有的源数据位），结果也可能与期望的不同。

因此，需要对数据进行类型转换，而不是将数据位从一个变量一对一映射到另一个变量。类型转换采用以下两种形式：

- 隐式转换： 从类型A到类型B的转换可在所有情况下进行，执行转换的规则非常简单，可以让编译器执行转换。
- 显式转换： 从类型A到类型B的转换只能在某些情况下进行，转换规则比较复杂，应进行某种类型的额外处理。

5.1.1 隐式转换

隐式转换不需要做任何工作，也不需要另外编写代码。考虑下面的代码：

```
var1 = var2;
```

如果var2的类型可以隐式地转换为var1的类型，这条赋值语句就涉及隐式转换。这两个变量的类型也可能相同，此时就不需要隐式转换。例如，ushort和char的值是可以互换的，因为它们都可以存储0和65 535之间的数字，在这两种类型之间可以进行隐式转换，如下面的代码所示：

```
ushort destinationVar;  
char sourceVar = 'a';  
destinationVar = sourceVar;  
WriteLine($"sourceVar val: {sourceVar}");  
WriteLine($"destinationVar val: {destinationVar}");
```

这里存储在sourceVar中的值放在destinationVar中。在用两个WriteLine()命令输出变量时，得到如下结果：

```
sourceVar val: a  
destinationVar val: 97
```

即使两个变量存储的信息相同，使用不同的类型解释它们时，方式也是不同的。

简单类型有许多隐式转换；bool和string没有隐式转换，但数值类型有一些隐式转换。表5-1列出了编译器可以隐式执行的数值转换（记住，char存储的是数值，所以char被当作数值类型）。

表5-1 隐式数值转换

--	--

类型	可以安全地转换为
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

不要担心——不需要记住这个表格，因为很容易看出编译器可以执行哪些隐式转换。第3章中的表3-1、表3-2和表3-3列出了每种简单数字类型的取值范围。这些类型的隐式转换规则是：任何类型A，只要其取值范围完全包含在类型B的取值范围内，就可以隐式转换为类型B。

其原因是很简单的。如果要把一个值放在变量中，而该值超出了变量的取值范围，就会出问题。例如，short类型的变量可以存储0-32 767的数字，而byte可以存储的最大值是255，所以如果要把short值转换为byte值，就会出问题。如果short包含的值在256和32 767之间，相应数值

就不能放在byte中。

但是，如果short类型变量中的值小于255，就应能转换这个值吗？答案是可以。具体地说，虽然可以，但必须使用显式转换。执行显式转换有点类似于“我已经知道你对我这么做提出了警告，但我将对其后果负责”。

5.1.2 显式转换

顾名思义，在明确要求编译器把数值从一种数据类型转换为另一种数据类型时，就是在执行显式转换。因此，这需要另外编写代码，代码的格式因转换方法而异。在学习显式转换代码前，首先分析如果不添加任何显式转换代码，会发生什么情况。

例如，下面对上一节的代码进行修改，试着把short值转换为byte类型：

```
byte destinationVar;
```

```
short sourceVar = 7;
```

```
destinationVar = sourceVar;
```

```
WriteLine($"sourceVar val: {sourceVar}");
```

```
WriteLine($"destinationVar val: {destinationVar}");
```

如果编译这段代码，就会产生如下错误：

```
Cannot implicitly convert type 'short' to 'byte'. An explicit  
(are you missing a cast?)
```

为成功编译这段代码，需要添加代码，进行显式转换。最简单的方式是把short变量强制转换为byte类型（由上述错误字符串提出）。强制转换就是强迫数据从一种类型转换为另一种类型，其语法比较简单：

```
(<destinationType  
  
>)<sourceVar  
  
>
```

这将会把<sourceVar>中的值转换为<destinationType>类型。

注意：这只在某些情况下是可行的。彼此之间几乎没有什么关系的类型或根本没有关系的类型不能进行强制转换。

因此可以使用这个语法修改示例，把short变量强制转换为byte类型：

```
byte destinationVar;  
short sourceVar = 7;  
destinationVar = (byte)
```

```
sourceVar;  
WriteLine($"sourceVar val: {sourceVar}");  
WriteLine($"destinationVar val: {destinationVar}");
```

得到如下结果：

```
sourceVar val: 7  
destinationVar val: 7
```

在试图把一个值转换为不兼容的变量类型时，会发生什么呢？以整数为例，不能把一个大整数放到一个太小的数值类型中。按如下所示修改代码就能证明这一点：

```
byte destinationVar;  
short sourceVar = 281  
  
;  
destinationVar = (byte)sourceVar;  
WriteLine($"sourceVar val: {sourceVar}");  
WriteLine($"destinationVar val: {destinationVar}");
```

结果如下：

```
sourceVar val: 281
destinationVar val: 25
```

发生了什么？看看这两个数字的二进制表示，以及可以存储在byte中的最大值255：

```
281 = 100011001
25 = 000011001
255 = 011111111
```

可以看出，源数据的最左边一位丢失了。这会引发一个问题：如何确定数据是何时丢失的？显然，当需要显式地把一种数据类型转换为另一种数据类型时，最好能够了解是否有数据丢失了。如果不知道这些，就会发生严重问题。例如，财务应用程序或确定火箭飞往月球的轨道的应用程序。

一种方式是检查源变量的值，将它与目标变量的取值范围进行比较。还有另一个技术，就是迫使系统特别注意运行期间的转换。在将一个值放在一个变量中时，如果该值过大，不能放在该类型的变量中，就会导致溢出，这就需要检查。

对于为表达式设置所谓的溢出检查上下文，需要用到两个关键字——checked和unchecked。按下述方式使用这两个关键字：

```
checked(<expression>
)
unchecked(<expression>
```


)

下面对上一个示例进行溢出检查：

```
byte destinationVar;  
short sourceVar = 281;  
destinationVar = checked(  
  
(byte)sourceVar)  
  
;  
WriteLine($"sourceVar val: {sourceVar}");  
WriteLine($"destinationVar val: {destinationVar}");
```

执行这段代码时，程序会崩溃，并显示如图5-1所示的错误信息（在OverflowCheck项目中编译这段代码）。

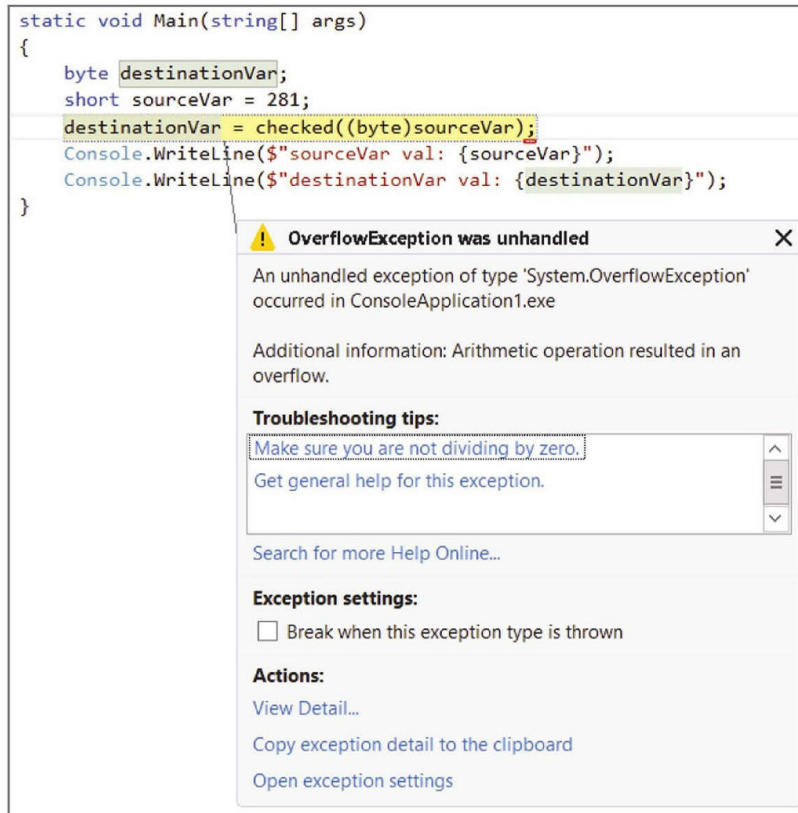


图5-1

但在这段代码中，如果用unchecked替代checked，就会得到与以前同样的结果，不会出现错误。这与前面的默认做法是一样的。

也可以配置应用程序，让这种类型的表达式都和包含checked关键字一样，除非表达式明确使用unchecked关键字（换言之，可以改变溢出检查的默认设置）。为此，应修改项目的属性：右击Solution Explorer窗口中的项目，选择Properties选项。单击窗口左边的Build，打开Build设置。

要修改的属性是一个Advanced设置，所以单击Advanced按钮。在打开的对话框中，选中Check for arithmetic overflow/underflow选项，如图5-2所示。默认情况下禁用这个设置，激活它可以提供上述checked行

为。

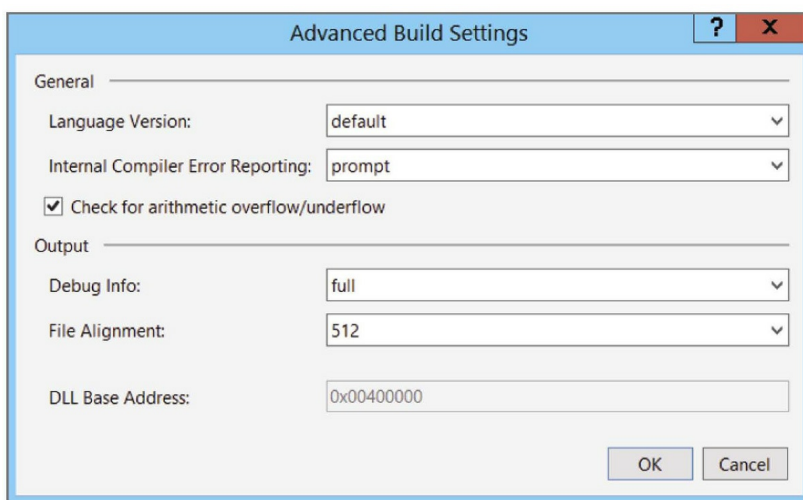


图5-2

5.1.3 使用**Convert**命令进行显式转换

前面章节中的许多“试一试”示例中使用的显式类型转换，与本章前面的示例有一些区别。前面使用`ToDouble()`等命令把字符串值转换为数值，显然，这种方式并不适用于所有字符串。

例如，如果使用`ToDouble()`把`Number`字符串转换为`double`值，在执行代码时，将看到如图5-3所示的对话框。

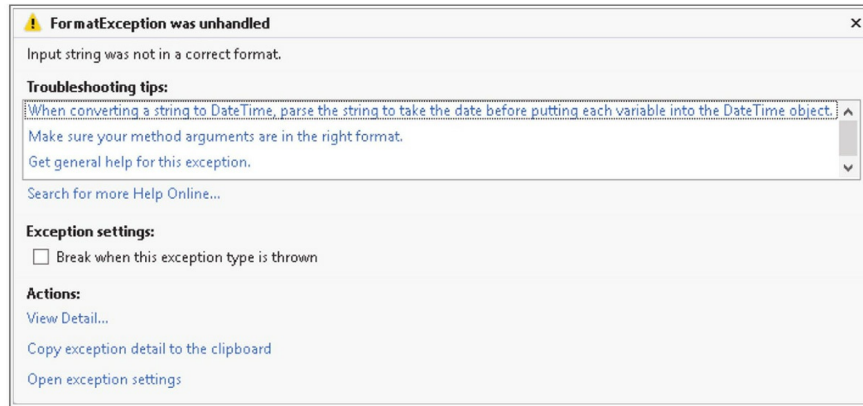


图5-3

可以看出，执行失败。为成功执行此类转换，所提供的字符串必须是数值的有效表达方式，该数还必须是不会溢出的数。数值的有效表达方式是：首先是一个可选符号（加号或减号），然后是0位或多位数字，一个可选的句点后跟一位或多位数字，接着是一个可选的e或E，后跟一个可选符号和一位或多位数字，除了还可能有空格（在这个序列之前或之后），不能有其他字符。利用这些可选的额外数据，可将-1.2451e-24这样复杂的字符串识别为数值。

对于这些转换要注意的一个问题是，它们总是要进行溢出检查，checked和unchecked关键字以及项目属性设置不起作用。

下面的示例包括本节介绍的许多转换类型。它声明和初始化许多不同类型的变量，再在它们之间进行隐式和显式转换。

试一试：类型转换实践：**Ch05Ex01\Program.cs**

(1) 在C:\BegVCSharp\Chapter05目录中创建一个新的控制台应用

程序Ch05Ex01。

(2) 把下述代码添加到Program.cs中：

```
static void Main(string[] args)
{
    short    shortResult, shortVal = 4;

    int      integerVal = 67;

    long     longResult;

    float    floatVal = 10.5F;

    double   doubleResult, doubleVal = 99.999;

    string   stringResult, stringVal = "17";
```

```
bool    boolVal = true;
```

```
WriteLine("Variable Conversion Examples\n");
```

```
doubleResult = floatVal * shortVal;
```

```
WriteLine($"Implicit, -> double: {floatVal} * {shortVal} -
```

```
shortResult = (short)floatVal;
```

```
WriteLine($"Explicit, -> short: {floatVal} -> {shortResult
```

```
stringResult = Convert.ToString(boolVal) +
```

```
Convert.ToString(doubleVal);
```

```
WriteLine($"Explicit, -> string: \"{boolVal}\" + \"{double
```

```
${stringResult}");
```

```
longResult = integerValue +.ToInt64(stringVal);
```

```
WriteLine($"Mixed, -> long: {integerVal} + {stringVal} ->
```

```
ReadKey();
```

}

(3) 执行代码，结果如图5-4所示。

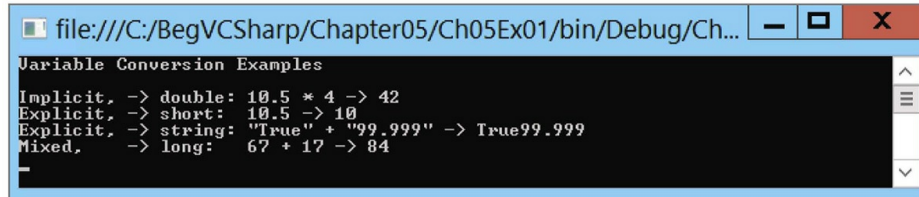


图5-4

示例说明

这个示例包含前面介绍的所有转换类型，既有像前面简短代码示例中的简单赋值，也有在表达式中进行的转换。必须考虑这两种情况，因为每个非一元运算符的处理都可能要进行类型转换，而不仅是赋值运算符。例如：

```
shortVal * floatVal
```

其中把一个short值与一个float值相乘。在这样的指令中，没有指定显式转换，所以如有可能，就会进行隐式转换。在这个示例中，唯一有意义的隐式转换是把short值转换为float值（因为把float值转换为short值需要进行显式转换），所以这里将使用隐式转换。

也可以覆盖这种行为，如下所示：

```
shortVal * (short)floatVal
```


注意： 有趣的是，两个short值相乘的结果并不会返回一个short值。因为这个操作的结果很可能大于32 767（这是short类型可以存储的最大值），所以这个操作的结果实际上是int值。

这个转换过程初看起来比较复杂，但只要按照运算符的优先级把表达式分解为不同的部分，就可以弄明白这个过程。

5.2 复杂的变量类型

除了这些简单的变量类型外，C#还提供了3个较复杂（但非常有用）的变量：枚举、结构和数组。

5.2.1 枚举

本书迄今介绍的每种类型（除string外）都有明确的取值范围。诚然，有些类型（如double）的取值范围非常大，可以看成是连续的，但它们仍是一个固定集合。最简单的示例是bool类型，它只能取两个值：true或false。

有时希望变量取的是一个固定集合中的值。例如，让orientation类型可以存储north、south、east或west中的一个值。

此时可以使用枚举类型。枚举可以完成这个orientation类型的任务：它们允许定义一个类型，其取值范围是用户提供的值的有限集合。所以，需要创建自己的枚举类型orientation，它可以从上述4个值中取一个值。

注意有一个附加步骤——不是仅声明一个给定类型的变量，而是声明和描述一个用户定义的类型，再声明这个新类型的变量。

定义枚举

可以用enum关键字定义枚举，如下所示：

```
enum<typeName  
  
>  
{  
    <value1  
  
>,  
    <value2  
  
>,  
    <value3  
  
>,  
    ...  
    <valueN  
  
>  
}
```

接着声明这个新类型的变量：

<typeName

><varName

>;

并赋值:

<varName

> =<typeName

>.<value

>;

枚举使用一个基本类型来存储。枚举类型可取的每个值都存储为该基本类型的一个值，默认情况下该类型为int。在枚举声明中添加类型，就可以指定其他基本类型:

enum<typeName

> : <underlyingType>

```
{  
    <value1
```

```
>,  
    <value2
```

```
>,  
    <value3
```

```
>,  
    ...  
    <valueN
```

```
>  
}
```

枚举的基本类型可以是byte、sbyte、short、ushort、int、uint、long和ulong。

默认情况下，每个值都会根据定义的顺序（从0开始），被自动赋予对应的基本类型值。这意味着<value1>的值是0，<value2>的值是1，<value3>的值是2，等等。可以重写这个赋值过程：使用=运算符，指定每个枚举的实际值：

```
enum<typeName  
  
> :<underlyingType>  
    {  
        <value1  
  
>  
        =<actualVal1>,  
  
  
        <value2  
  
>  
        =<actualVal2>,
```

<value3

>

=<actualVal3>,

...

<valueN

>

=<actualValN>

```
}
```

还可以使用一个值作为另一个枚举的基础值，为多个枚举指定相同的值：

```
enum<typeName
```

```
> :<underlyingType
```

```
>
```

```
{
```

```
<value1
```

```
> = <actualVal1
```

```
> ,
```

```
<value2
```

```
> = <
```

```
value1
```



```
>,
```

```
<value3
```

```
>,
```

```
...
```

```
<valueN
```

```
> = <actualValN
```

```
>
```

```
}
```

未赋值的任何值都会自动获得一个初始值，这里使用的值是从比上一个明确声明的值大1开始的序列。例如，在上面的代码中，<value3 >的值是<value1 >+1。

注意这可能会产生预料不到的问题，在一个定义（如<value2 >=

<value1 >) 后指定的值可能与其他值相同。例如，在下面的代码中，<value4 >的值与<value2 >的值相同：

```
enum <typeName>
```

```
: <underlyingType>
```

```
{  
    <value1> = <actualVal1>,
```

```
<value2>,
```

```
    <value3> = <value1>,
```

```
<value4>,
```

```
...
```

```
<valueN> = <actualValN>
```

```
}
```

当然，如果这正是希望的结果，代码就是正确的。还要注意，以循环方式赋值可能会产生错误，例如：

```
enum <typeName>
```

```
: <underlyingType>
```

```
{  
    <value1> = <value2>,
```

```
    <value2> = <value1>
```

```
}
```

下面看一个示例。其代码定义了一个枚举orientation，然后演示了它的用法。

试一试：使用枚举：**Ch05Ex02\Program.cs**

(1) 在C:\BegVCSharp\Chapter05目录中创建一个新的控制台应用程序Ch05Ex02。

(2) 把下列代码添加到Program.cs中：

```
namespace Ch05Ex02
{
    enum orientation : byte

    {

        north = 1,

        south = 2,

        east = 3,

        west = 4

    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        orientation myDirection = orientation.north;

        WriteLine($"myDirection = {myDirection}");

        ReadKey();

    }
}
```

(3) 运行应用程序，应得到如图5-5所示的输出结果。

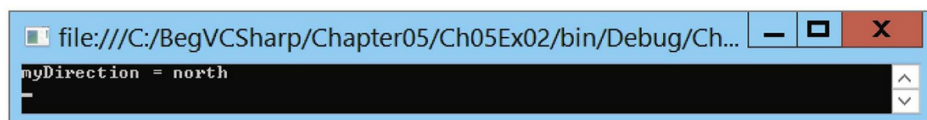


图5-5

(4) 退出应用程序，修改代码，如下所示：

```
byte directionByte;
```

```
string directionString;
```

```
orientation myDirection = orientation.north;
```

```
WriteLine($"myDirection = {myDirection}");
```

```
directionByte = (byte)myDirection;
```

```
directionString = Convert.ToString(myDirection);
```

```
WriteLine($"byte equivalent = {directionByte}");
```

```
WriteLine($"string equivalent = {directionString}");
```

```
ReadKey();
```

(5) 再次运行应用程序，输出结果如图5-6所示。

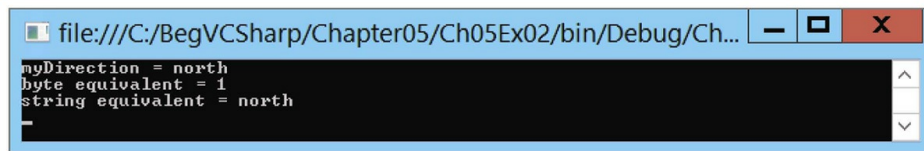


图5-6

示例说明

这段代码定义并使用了一个枚举类型`orientation`。首先要注意，类型定义代码放在名称空间`Ch05Ex02`中，但没有与其余代码放在一起。这是因为在运行期间，定义代码并不像执行应用程序中的代码那样一行一行地执行。应用程序是从我们熟悉的位置开始执行的，它可以访问新类型，因为该类型位于同一个名称空间中。

这个示例的第一个迭代演示了创建新类型的变量，给它赋值以及把它输出到屏幕上的基本方法。接着修改代码，把枚举值转换为其他类型。注意这里必须使用显式转换。即使`orientation`的基本类型是`byte`，也仍必须使用`(byte)`强制实现类型转换，把`myDirection`的值转换为`byte`类型：

```
directionByte = (byte)myDirection;
```

如果要将`byte`类型转换为`orientation`，也同样需要进行显式转换。例

如，可以使用下述代码将byte变量myByte转换为orientation值，并将这个值赋给myDirection：

```
myDirection = (orientation)myByte;
```

当然，这里必须小心，因为并不是所有byte类型变量的值都可以映射为已定义的orientation值。orientation类型可以存储其他byte值，所以这么做不会直接产生一个错误，但会在应用程序的后面违反逻辑。

要获得枚举的字符串值，可以使用Convert.ToString()：

```
directionString = Convert.ToString(myDirection);
```

使用（string）强制类型转换是行不通的，因为需要进行的处理并不仅是把存储在枚举变量中的数据放在string变量中，而是更复杂一些。另外，可以使用变量本身的ToString()命令。下面的代码与使用Convert.ToString()的效果相同：

```
directionString = myDirection.ToString();
```

也可以把string转换为枚举值，但其语法稍复杂一些。有一个特定命令用于完成此类转换，即Enum.Parse()，其用法如下：

```
(enumerationType)Enum.Parse(typeof(enumerationType), enumerati
```

这里使用了另一个运算符typeof，它可以得到操作数的类型。对orientation类型使用这个命令，如下所示：

```
string myString = "north";  
orientation myDirection = (orientation)Enum.Parse(typeof(orien
```

当然，并非所有字符串值都会映射为一个orientation值。如果传送的一个值不能映射为枚举值中的一个，就会产生错误。与C#中的其他值一样，这些值是区分大小写的，所以如果字符串与一个值相同，但大小写不同（例如，将myString设置为North而不是north），就会产生错误。

5.2.2 结构

下一个要介绍的变量类型是结构（**struct**，**structure**的简写）。结构就是由几个数据组成的数据结构，这些数据可能具有不同的类型。根据这个结构，可以定义自己的变量类型。例如，假定要存储从起点开始到某一位置的路径，路径由方向和距离值（英里）组成。为简单起见，假定该方向是指南针上的一点（这样，方向就可以用上一节的orientation枚举来表示），距离值可以用double类型来表示。

通过前面的代码，可用两个不同的变量来表示路径：

```
orientation  myDirection;  
double       myDistance;
```

像这样使用两个变量，是没有错误的，但在一个地方存储这些信息更加简单（在需要多个路径时，就尤为简单）。

定义结构

使用**struct**关键字定义结构，如下所示：

```

struct<typeName

>

{
    <memberDeclarations

>

}

```

<memberDeclarations>部分包含变量的声明（称为结构的数据成员），其格式与前面的变量声明一样。每个成员的声明都采用如下形式：

```

    <accessibility

><type

><name

>;

```

要让调用结构的代码访问该结构的数据成员，可以对<accessibility>使用关键字public，例如：

```
struct route
{
    public orientation    direction;
    public double         distance;
}
```

定义结构类型后，就可以定义该结构类型的变量：

```
route myRoute;
```

还可以通过句点字符访问这个组合变量中的数据成员：

```
myRoute.direction = orientation.north;
myRoute.distance = 2.5;
```

把这个类型放在下面的“试一试”示例中，其中使用上一个“试一试”示例中的orientation枚举和上面的route结构。本例在代码中处理这个结构，以便了解结构的工作原理。

试一试：使用结构：Ch05Ex03\Program.cs

(1) 在C:\BegVCSharp\Chapter05目录中创建一个新的控制台应用程序Ch05Ex03。

(2) 将下列代码添加到Program.cs中：

```
namespace Ch05Ex03
```

```
{  
    enum orientation: byte
```

```
{
```

```
    north = 1,
```

```
    south = 2,
```

```
    east = 3,
```

```
    west = 4
```

```
}
```

```
struct route
```

```
{
```

```
public orientation direction;
```

```
public double distance;
```

```
}
```

```
class Program
```

```
{
```

```
static void Main(string[] args)
```

```
{
```

```
route myRoute;
```

```
int myDirection = -1;
```

```
double myDistance;
```

```
WriteLine("1) North\n2) South\n3) East\n4) West");
```

```
do
```

```
{
```

```
WriteLine("Select a direction:");
```

```
myDirection =.ToInt32(ReadLine());
```

```
}
```

```
while ((myDirection<1) || (myDirection > 4));
```

```
WriteLine("Input a distance:");
```

```
myDistance = ToDouble(ReadLine());
```

```
myRoute.direction = (orientation)myDirection;
```

```
myRoute.distance = myDistance;
```



```
WriteLine($"myRoute specifies a direction of {myRoute.d
```

```
$"and a distance of {myRoute.distance}");
```

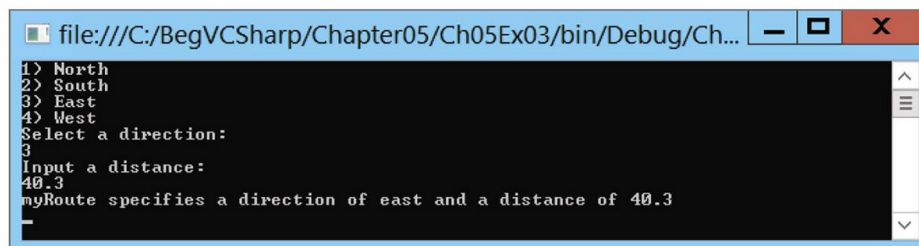
```
ReadKey();
```

```
}
```

```
}
```

```
}
```

（3）执行代码，输入一个介于1和4之间的数字，以选择一个方向，输入一个距离值，结果如图5-7所示。



```
file:///C:/BegVCSharp/Chapter05/Ch05Ex03/bin/Debug/Ch...
1> North
2> South
3> East
4> West
Select a direction:
3
Input a distance:
40.3
myRoute specifies a direction of east and a distance of 40.3
```

图5-7

示例说明

结构和枚举一样，也是在代码的主体之外声明的。在名称空间声明中声明route结构及其使用的orientation枚举：

```
enum orientation: byte
{
    north = 1,
    south = 2,
    east = 3,
    west = 4
}
struct route
{
    public orientation    direction;
    public double         distance;
}
```

代码的主体结构与前面的一些示例代码类似，要求用户输入一些信息，并显示它们。把方向选择放在do循环中，对用户的输入进行有效性检查，拒绝不属于1-4范围的整数输入（选择该范围中的值可以映射到枚举成员，从而方便赋值）。

注意： 不能解释为整数的输入会导致一个错误。本章后面会说明其原因和处理方法。

注意，在引用route的成员时，处理它们的方式与处理成员类型相同的变量完全一样。赋值语句如下所示：

```
myRoute.direction = (orientation)myDirection;  
myRoute.distance = myDistance;
```

可直接把输入的值放到`myRoute.distance`中，而不会有负面效果，如下所示：

```
myRoute.distance = ToDouble(ReadLine());
```

还应进行有效性验证，但这段代码不存在这一步骤。对结构成员的任何访问都以相同的方式处理。`<structVar>.<memberVar>`形式的表达式可计算`<memberVar>`类型的变量。

5.2.3 数组

前面的所有类型有一个共同点：它们都只存储一个值（结构中存储一组值）。有时，需要存储许多数据，这样就会带来不便。有时需要同时存储几个类型相同的值，而不想为每个值使用不同的变量。

例如，假定要对所有朋友的姓名执行一些操作。可以使用简单的字符串变量，如下所示：

```
string friendName1 = "Todd Anthony";  
string friendName2 = "Kevin Holton";  
string friendName3 = "Shane Laigle";
```

但这看起来需要做很多工作，特别是需要编写不同的代码来处理每个变量。例如，不能在循环中迭代这个字符串列表。

另一种方式是使用数组。数组是一个变量的索引列表，存储在数组类型的变量中。例如，有一个数组friendNames存储上述3个名字。在方括号中指定索引，即可访问该数组中的各个成员，如下所示：

```
friendNames[<index>  
  
]
```

这个索引是一个整数，第一个条目的索引是0，第二个条目的索引是1，依此类推。这样就可以使用循环遍历所有条目，例如：

```
int i;  
for (i = 0; i<3; i++)  
{  
    WriteLine($"Name with index of {i}: {friendNames[i]}");  
}
```

数组有一个基本类型，数组中的各个条目都是这种类型。friendNames数组的基本类型是字符串，因为它要存储string变量。数组的条目通常称为元素。

1. 声明数组

以下述方式声明数组：

```
<baseType
```

```
>[]<name
```

```
>;
```

其中，<*baseType*>可以是任何变量类型，包括本章前面介绍的枚举和结构类型。数组必须在访问之前初始化，不能像下面这样访问数组或给数组元素赋值：

```
int[] myIntArray;  
myIntArray[10] = 5;
```

数组的初始化有两种方式。可以字面值形式指定数组的完整内容，也可以指定数组的大小，再使用关键字**new**初始化所有数组元素。

要使用字面值指定数组，只需提供一个用逗号分隔的元素值列表，该列表放在花括号中，例如：

```
int[] myIntArray = { 5, 9, 10, 2, 99 };
```

其中，**myIntArray**有5个元素，每个元素都被赋予一个整数值。

另一种方式需要使用下述语法：

```
int[] myIntArray = new int[5];
```

这里使用关键字**new**显式地初始化数组，用一个常量值定义其大小。这种方式会给所有数组元素赋予同一个默认值，对于数值类型来说，其默认值是0。也可以使用非常量的变量来进行初始化，例如：

```
int[] myIntArray = new int[arraySize];
```

还可以组合使用这两种初始化方式：

```
int[] myIntArray = new int[5] { 5, 9, 10, 2, 99 };
```

使用这种方式，数组大小必须与元素个数相匹配。例如，不能编写如下代码：

```
int[] myIntArray = new int[10] { 5, 9, 10, 2, 99 };
```

其中数组定义为有10个元素，但只定义了5个元素，所以编译会失败。如果使用变量定义其大小，该变量必须是一个常量，例如：

```
const int arraySize = 5;  
int[] myIntArray = new int[arraySize] { 5, 9, 10, 2, 99 };
```

如果省略了关键字const，运行这段代码就会失败。

与其他变量类型一样，并非必须在声明数组的代码行中初始化该数组。下面的代码是合法的：

```
int[] myIntArray;  
myIntArray = new int[5];
```

下面的“试一试”示例利用了本节开头的示例，创建并使用一个字符串数组。

试一试：使用数组： **Ch05Ex04Program.cs**

(1) 在C:\BegVCSharp\Chapter05目录中创建一个新的控制台应用程序Ch05Ex04。

(2) 将下列代码添加到Program.cs中:

```
static void Main(string[] args)
{
    string[] friendNames = { "Todd Anthony", "Kevin Holton",

                               "Shane Laigle" };

    int i;

    WriteLine($"Here are {friendNames.Length} of my friends:")

    for (i = 0; i<friendNames.Length; i++)
```

```
{
```

```
    WriteLine(friendNames[i]);
```

```
}
```

```
ReadKey();
```

```
}
```

(3) 执行代码，结果如图5-8所示。

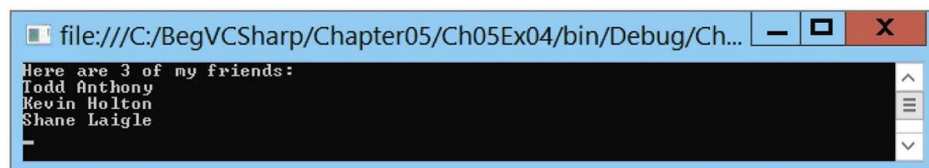


图5-8

示例说明

这段代码用3个值建立了一个string数组，并在for循环中把它们列在控制台上。使用friendNames.Length来确定数组中的元素个数：

```
WriteLine($"Here are {friendNames.Length} of my friends:");
```

这是获取数组大小的简便方法。在for循环中输出值容易出错。例如，把<改为<=，如下所示：

```
for (i = 0; i<= friendNames.Length; i++)  
{  
    WriteLine(friendNames[i]);  
}
```

编译并执行上述代码，就会弹出如图5-9所示的对话框。

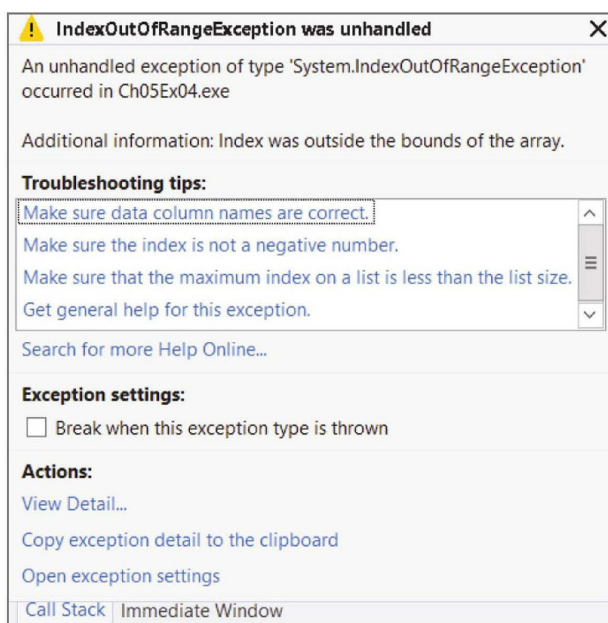


图5-9

这里，代码试图访问friendNames[3]。记住，数组索引从0开始，所

以最后一个元素是`friendNames[2]`。如果试图访问超出数组大小的元素，代码就会出问题。还可以通过一个更具弹性的方法来访问数组的所有成员，即使用`foreach`循环。

2. `foreach`循环

`foreach`循环可以使用一种简便的语法来定位数组中的每个元素：

```
foreach (<baseType  
  
><name  
  
> in<array  
  
>)  
    {  
        // can use<name  
  
> for each element  
    }
```

这个循环会迭代每个元素，依次把每个元素放在变量`<name >`中，

且不存在访问非法元素的危险。不需要考虑数组中有多少个元素，并确保将在循环中使用每个元素。使用这个循环，可以修改上个示例中的代码，如下所示：

```
static void Main(string[] args)
{
    string[] friendNames = { "Todd Anthony", "Kevin Holton",
                             "Shane Laigle" };
    WriteLine($"Here are {friendNames.Length} of my friends:")
    foreach (string friendName in friendNames)

    {

        WriteLine(friendName);

    }

    ReadKey();
}
```

这段代码的输出结果与前面的“试一试”示例完全相同。使用这种方法 and 标准的for循环的主要区别在于：foreach循环对数组内容进行只读访问，所以不能改变任何元素的值。例如，不能编写如下代码：

```
foreach (string friendName in friendNames)
{
    friendName = "Rupert the bear";
}
```

如果编译这段代码，就会失败。但如果使用简单的for循环，就可以给数组元素赋值。

3. 多维数组

多维数组是使用多个索引访问其元素的数组。例如，假定要确定一座山相对于某位置的高度，可使用两个坐标x和y来指定一个位置。把这两个坐标用作索引，让数组hillHeight可以用每对坐标来存储高度，这就要使用多维数组了。

像这样的二维数组可以声明如下：

```
<baseType  
  
>[,]<name
```

```
>;
```

多维数组只需要更多逗号，例如：

```
<baseType
```

```
>[, , , ]<name
```

```
>;
```

该语句声明了一个4维数组。赋值也使用类似的语法，用逗号分隔大小。要声明和初始化二维数组`hillHeight`，其基本类型是`double`，`x`的大小是3，`y`的大小是4，则需要：

```
double[,] hillHeight = new double[3,4];
```

还可以使用字面值进行初始赋值。这里使用嵌套的花括号块，它们之间用逗号分开，例如：

```
double[,] hillHeight = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3,
```

这个数组的维度与前面的相同，也是3行4列。通过提供字面值隐式定义了这些维度。

要访问多维数组中的每个元素，只需要指定它们的索引，并用逗号分开，例如：

hillHeight[2,1]

接着就可以像其他元素那样处理它了。这个表达式将访问上面定义的第3个嵌套数组中的第2个元素（其值是4）。记住，索引从0开始，第一个数字是嵌套的数组。换言之，第一个数字指定花括号对，第2个数字指定该对花括号中的元素。用图5-10来表示这个数组。

hillHeight [0,0] 1	hillHeight [0,1] 2	hillHeight [0,2] 3	hillHeight [0,3] 4
hillHeight [1,0] 2	hillHeight [1,1] 3	hillHeight [1,2] 4	hillHeight [1,3] 5
hillHeight [2,0] 3	hillHeight [2,1] 4	hillHeight [2,2] 5	hillHeight [2,3] 6

图5-10

foreach循环可以访问多维数组中的所有元素，其方式与访问一维数组相同，例如：

```
double[,] hillHeight = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3,  
foreach (double height in hillHeight)  
{  
    WriteLine("{0}", height);  
}
```

元素的输出顺序与赋予字面值的顺序相同（这里显示了元素的标识符而非实际值）：

```
hillHeight[0,0]
hillHeight[0,1]
hillHeight[0,2]
hillHeight[0,3]
hillHeight[1,0]
hillHeight[1,1]
hillHeight[1,2]
...
```

4. 数组的数组

上一节讨论的多维数组可称为矩形数组，这是因为每一行的元素个数都相同。使用上一个示例，任何一个x坐标都可以对应0至3的y坐标。

也可以使用锯齿数组（jagged array），其中每行的元素个数可能不同。为此，需要有这样一个数组，其中的每个元素都是另一个数组。也可以有数组的数组的数组，甚至更复杂的数组。但是，注意这些数组都必须有相同的基本类型。

声明数组的数组时，其语法要求在数组的声明中指定多个方括号对，例如：

```
int[][] jaggedIntArray;
```

但初始化这样的数组不像初始化多维数组那样简单，例如不能采用以下声明方式：

```
jaggedIntArray = new int[3][4];
```

即使这样做了，也不是很有效，因为使用简单的多维数组可以较为轻松地取得相同的结果。也不能使用下面的代码：

```
jaggedIntArray = { { 1, 2, 3 }, { 1 }, { 1, 2 } };
```

有两种方式：可以初始化包含其他数组的数组（为清晰起见，称其为子数组），然后依次初始化子数组。

```
jaggedIntArray = new int[2][];  
jaggedIntArray[0] = new int[3];  
jaggedIntArray[1] = new int[4];
```

也可以使用上述字面值赋值的一种改进形式：

```
jaggedIntArray = new int[3][] { new int[] { 1, 2, 3 }, new int  
                                new int[] { 1, 2 } };
```

也可以进行简化，把数组的初始化和声明放在同一行上，如下所示：

```
int[][] jaggedIntArray = { new int[] { 1, 2, 3 }, new int[] {  
                            new int[] { 1, 2 } };
```

可以对锯齿数组使用foreach循环，但通常需要使用嵌套的foreach循环才能得到实际数据。例如，假定下述锯齿数组包含10个数组，每个数组又包含一个整数数组，其元素是1-10的约数：

```
int[][] divisors1To10 = { new int[] { 1 },  
                           new int[] { 1, 2 },  
                           new int[] { 1, 3 },
```



```
new int[] { 1, 2, 4 },
new int[] { 1, 5 },
new int[] { 1, 2, 3, 6 },
new int[] { 1, 7 },
new int[] { 1, 2, 4, 8 },
new int[] { 1, 3, 9 },
new int[] { 1, 2, 5, 10 } };
```

下面的代码会失败：

```
foreach (int divisor in divisors1To10)
{
    WriteLine(divisor);
}
```

这是因为数组divisors1To10包含int[]元素而不是int元素。正确的做法是循环遍历每个子数组和数组本身：

```
foreach (int[] divisorsOfInt in divisors1To10)
{
    foreach(int divisor in divisorsOfInt)
    {
        WriteLine(divisor);
    }
}
```

可以看出，使用锯齿数组的语法要复杂得多！大多数情况下，使用矩形数组比较简单，这是一种比较简单的存储方式。但是，有时必须使

用锯齿数组，所以知道怎么使用它们是没有坏处的。一个例子是，使用XML文档，其中一些元素有子元素，而一些元素没有。

5.3 字符串的处理

到目前为止，对字符串的使用还仅限于把字符串写到控制台，从控制台读取字符串，以及使用+运算符连接字符串。在编写较有趣的应用程序时，会发现字符串的操作非常多。所以，下面占用几页的篇幅介绍C#中较常用的字符串处理技巧。

首先要注意，`string`类型的变量可以看成`char`变量的只读数组。这样，就可以使用下面的语法访问每个字符：

```
string myString = "A string";  
char myChar = myString[1];
```

但不能采用这种方式为各个字符赋值。为获得一个可写的`char`数组，可以使用下面的代码，其中使用了数组变量的`ToCharArray()`命令：

```
string myString = "A string";  
char[] myChars = myString.ToCharArray();
```

接着就可以采用标准方式处理`char`数组了。也可在`foreach`循环中使用字符串，例如：

```
foreach (char character in myString)  
{  
    WriteLine($"{character}");  
}
```

与数组一样，还可以使用`myString.Length`获取元素个数，这将给出字符串中的字符数，例如：

```
string myString = ReadLine();  
WriteLine($"You typed {myString.Length} characters.");
```

其他基本字符串处理技巧采用与这个`<string>.ToCharArray()`命令类似的格式使用命令。两个简单却有效的命令是`<string>.ToLower()`和`<string>.ToUpper()`。它们可以分别把字符串转换为小写或大写形式。为理解它们的重要作用，可以考虑下面的情形：要检查用户的某个响应，例如字符串`yes`。如果可以把用户输入的字符串转换为小写形式，就也能检查字符串`YES`、`Yes`、`yeS`等，第4章介绍了这样一个示例：

```
string userResponse = ReadLine();  
if (userResponse.ToLower() == "yes")  
{  
    // Act on response.  
}
```

注意，这个命令与本节的其他命令一样，并未真正改变应用它的字符串。把这个命令与字符串结合使用，就会创建一个新的字符串，以便与另一个字符串进行比较（如上所述），或者赋给另一个变量。该变量可以是当前操作的变量，例如：

```
userResponse = userResponse.ToLower();
```

记住这一点很重要，因为只写出下面的代码是没用的：

```
userResponse.ToLower();
```

下面看看在简化用户输入方面还可以做什么。如果用户无意间在输入内容的前面或后面添加了多余的空格，会怎样？此时，上述代码就不起作用了。这就需要删除输入字符串前后的空格，此时可以使用`<string>.Trim()`命令来处理：

```
string userResponse = ReadLine();
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
{
    // Act on response.
}
```

使用该命令，还可以检测如下字符串：

```
" YES"
"Yes "
```

也可以使用这些命令删除其他字符，只要在一个`char`数组中指定这些字符即可，例如：

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = ReadLine();
userResponse = userResponse.ToLower();
userResponse = userResponse.Trim(trimChars);
if (userResponse == "y")
{
    // Act on response.
}
```

这将删除字符串前后的所有空格、字母e和s。如果字符串中没有其他字符，就会检测以下字符串：

```
"Yeeeeees"
```

```
" y"
```

还可以使用`<string >.TrimStart()`和`<string >.TrimEnd()`命令，它们可以把字符串前面或后面的空格删掉。使用这些命令时也可以指定char数组。

还有另外两个字符串命令可以处理字符串的空格：`<string >.PadLeft()`和`<string >.PadRight()`。它们可以在字符串的左边或右边添加空格，使字符串达到指定的长度。其语法如下：

```
<string  
  
>.PadX(<desiredLength  
  
>);
```

例如：

```
myString = "Aligned";  
myString = myString.PadLeft(10);
```

这将在myString中把3个空格添加到单词Aligned的左边。这些方法可以用于在列中对齐字符串，特别适于放置包含数字的字符串。

与修整命令一样，还可以按第二种方式使用这些命令，即提供要添加到字符串上的字符。但是这需要一个char字符，而不是像修整命令那样指定一个char数组。例如：

```
myString = "Aligned";  
myString = myString.PadLeft(10, '-');
```

这将在myString的开头加上3个短横线。

还有许多这样的字符串处理命令，其中一些只用于非常特殊的情况，在后面的章节中遇到它们时再进行讨论。在继续下面的内容之前，有必要介绍Visual Studio 2015中的一个特性，前几章曾提及（特别是本章）这个特性。下面的示例会试验语句自动完成功能，IDE通过这种功能给出用户有可能要插入的代码。

试一试：VS中的语句自动完成功能：Ch05Ex05\Program.cs

（1）在C:\BegVCSharp\Chapter05目录中创建一个新的控制台应用程序Ch05Ex05。

（2）在Program.cs中输入下列代码，注意输入过程中弹出的窗口：

```
static void Main(string[] args)  
{  
    string myString = "This is a test.";
```

```
char[] separator = {' '};
```

```
string[] myWords;
```

```
myWords = myString.
```

```
}
```

(3) 输入最后的句点时，注意会弹出如图5-11所示的窗口。

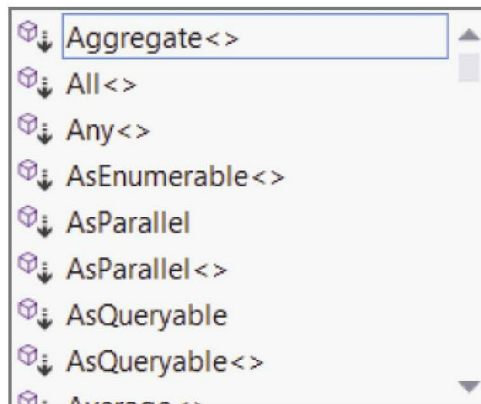


图5-11

(4) 不要移动光标，键入sp，弹出窗口就会改变，显示一个工具

提示窗口，如图5-12所示。

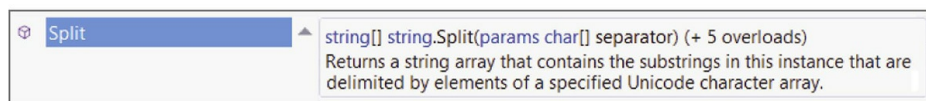


图5-12

(5) 输入字符“(se”，会弹出另一个窗口和工具提示，如图5-13所示。



图5-13

(6) 输入两个字符“);”，代码如下所示，弹出窗口随之消失：

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = {' '};
    string[] myWords;
    myWords = myString.Split(separator);
}
```

```
}
```

(7) 添加下述代码，注意弹出的窗口：

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = {' '};
    string[] myWords;
    myWords = myString.Split(separator);
    foreach (string word in myWords)

    {

        WriteLine($"{word}");

    }

    ReadKey();
}
```

```
}
```

(8) 执行代码，结果如图5-14所示。

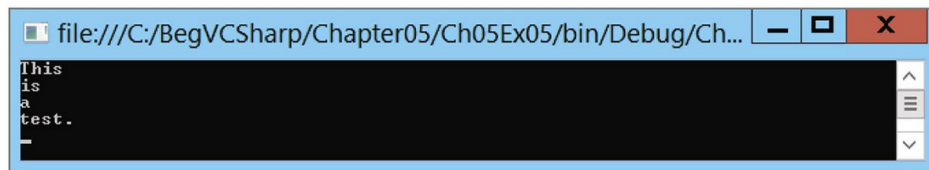


图5-14

示例说明

在这段代码中，要注意两点。第一点是所使用的新字符串命令，第二点是使用了自动完成功能。使用命令`<string>.Split()`把string字符串转换为string数组，把它在指定的位置隔开。这些位置采用char数组形式，在本例中该数组只有一个元素，即空格字符：

```
char[] separator = {' '};
```

下面的代码把字符串在每个空格处分解开，并获取得到的子字符串，即得到包含单独单词的数组：

```
string[] myWords;  
myWords = myString.Split(separator);
```

接着使用foreach循环迭代这个数组中的单词，并把这些单词写到控制台：

```
foreach (string word in myWords)
{
    WriteLine($"{word}");
}
```

注意： 得到的每个单词都没有空格，单词的内部和两端都没有空格。在使用split()时，删除了分隔符。

5.4 练习

(1) 下面的转换哪些不是隐式转换？

- a. int转换为short
- b. short转换为int
- c. bool转换为string
- d. byte转换为float

(2) 以short类型作为基本类型编写一个color枚举，使其包含彩虹的颜色加上黑色和白色。这个枚举可使用byte类型作为基本类型吗？

(3) 下面的代码可以成功编译吗？为什么？

```
string[] blab = new string[5]  
blab[5] = 5th string.
```

(4) 编写一个控制台应用程序，它接收用户输入的一个字符串，将其中的字符以与输入相反的顺序输出。

(5) 编写一个控制台应用程序，它接收一个字符串，用yes替换字符串中所有的no。

(6) 编写一个控制台应用程序，给字符串中的每个单词加上双引号。

附录A给出了练习答案。

5.5 本章要点

主题	要点
类型转换	值可以从一种类型转换为另一种类型，但在转换时应遵循一些规则。隐式转换是自动进行的，但只有当源值类型的所有可能值都可以在目标值类型中使用时，才能进行隐式转换。也可以进行显式转换，但可能得不到期望的值，甚至可能出错
枚举	枚举是包含一组离散值的类型，每个离散值都有一个名称。枚举用 enum 关键字定义，以便在代码中理解它们，因为它们的可读性都很高。枚举有基本的数值类型（默认是 int ），可使用枚举值的这个属性在枚举值和数值之间转换，或者标识枚举值
结构	结构是同时包含几个不同值的类型。结构用 struct 关键字定义。包含在结构中的每个值都有名称和类型，存储在结构中的每个值的类型不一定相同
数组	数组是同类型数值的集合。数组有固定的大小或长度，确定了数组可以包含多少个值。可以定义多维数组或锯齿数组来保存不同数量和形状的数据。还可以使用 foreach 循环来迭代数组中的值

第6章 函数

本章内容：

- 如何定义和使用既不接受任何数据也不返回任何数据的简单函数
- 如何在函数中传入传出数据
- 使用变量作用域
- 如何结合使用Main()函数和命令行参数
- 如何把函数提供为结构类型的成员
- 如何使用函数重载
- 如何使用委托

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 6 Code后，可以找到与本章示例对应的单独文件。

我们迄今看到的代码都是以单个代码块的形式出现的，其中包含一些重复执行的循环代码，以及有条件地执行的分支语句。如果要对数据执行某种操作，就应把所需要的代码放在合适的地方。

这种代码结构的作用是有限的。某些任务常需要在一个程序中执行

好几次，例如查找数组中的最大值。此时可以把相同（或几乎相同）的代码块按照需要放在应用程序中，但这样做存在一个问题。在某个常见任务中，即使进行非常小的改动（例如，修改某个代码错误），也需要修改多个代码块，而这些代码块可能分布在整个应用程序中。如果忘了修改其中一个代码块，就会产生很大影响，导致整个应用程序失败。另外，应用程序也较长。

解决这个问题的方法是使用函数。在C#中，函数可提供在应用程序中的任何一处执行的代码块。

注意：本章介绍的特定类型的函数称为“方法”。但是，这个术语在.NET编程中有非常特殊的含义，本书后面会详细讨论它，所以现在不使用这个术语。

例如，有一个函数返回数组中的最大值，可在代码的任何位置使用这个函数，且在每个地方都使用相同的代码行。因为只需要提供一次这段代码，所以对代码的任何修改将影响使用该函数进行的计算。这个函数可以看成包含可重用的代码。

函数还可以提高代码的可读性，因为可以使用函数将相关代码组合在一起。这样，应用程序主体就会非常短，因为代码的内部工作被分散了。这类似于在IDE中使用大纲视图将代码区域折叠在一起，应用程序的结构更加合理。

函数还可以用于创建多用途的代码，让它们对不同的数据执行相同的操作。可以采用参数形式为函数提供信息，以返回值的形式得到函数

的结果。在上面的示例中，参数就是一个要搜索的数组，而返回值就是数组中的最大值。这意味着每次可以使用同一函数处理不同的数组。函数的定义包括函数名、返回类型以及一个参数列表，这个参数列表指定了该函数需要的参数数量和参数类型。函数的名称和参数（不是返回类型）共同定义了函数的签名。

6.1 定义和使用函数

本节介绍如何将函数添加到应用程序中，以及如何在代码中使用（调用）它们。首先从基础知识开始，看看不与调用代码交换任何数据的简单函数，然后介绍更高级的函数用法。首先分析一个示例。

试一试：定义和使用基本函数：**Ch06Ex01\Program.cs**

（1）在C:\BegVCSharp\Chapter06目录中创建一个新的控制台应用程序Ch06Ex01。

（2）把下述代码添加到Program.cs中：

```
class Program
{
    static void Write()

{

    WriteLine("Text output from function.");
```

```
}

static void Main(string[] args)
{
    Write();

    ReadKey();

}
}
```

(3) 执行代码，结果如图6-1所示。

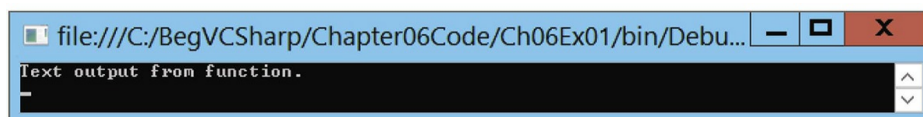


图6-1

示例说明

下面的4行代码定义了函数Write():

```
static void Write()  
{  
    WriteLine("  
  
Text output from function."  
  
);  
}
```

这些代码把一些文本输出到控制台窗口中。但此时这些并不重要，我们更关心定义和使用函数的机制。

函数定义由以下几部分组成：

- 两个关键字：static和void
- 函数名后跟圆括号，如Write()
- 一个要执行的代码块，放在花括号中

注意： 一般采用PascalCase形式编写函数名。

定义Write()函数的代码非常类似于应用程序中的其他代码：

```
static void Main(string[] args)
```

```
{  
    ...  
}
```

这是因为，到目前为止我们编写的所有代码（类型定义除外）都是函数的一部分。函数Main()是控制台应用程序的入口点函数。当执行C#应用程序时，就会调用它包含的入口点函数，这个函数执行完毕后，应用程序就终止了。所有C#可执行代码都必须有一个入口点。

Main()函数和Write()函数的唯一区别（除了它们包含的代码）是函数名Main后面的圆括号中还有一些代码，这是指定参数的方式，详见后面的内容。

如上所述，Main()函数和Write()函数都是使用关键字static和void定义的。关键字static与面向对象的概念相关，本书在后面讨论。现在只需要记住，本节的应用程序中所使用的所有函数都必须使用这个关键字。

void更容易解释。这个关键字表明函数没有返回值。本章后面将讨论函数有返回值时需要编写什么代码。

继续下去，调用函数的代码如下所示：

```
Write();
```

键入函数名，后跟空括号即可。当程序执行到这行代码时，就会运行Write()函数中的代码。

注意： 在定义和调用函数时，必须使用圆括号。如果删除它们，

将无法编译代码。

6.1.1 返回值

通过函数进行数据交换的最简单方式是利用返回值。有返回值的函数会最终计算得到这个值，就像在表达式中使用变量时，会计算得到变量包含的值一样。与变量一样，返回值也有数据类型。

例如，有一个函数`GetString()`，其返回值是一个字符串，可以在代码中使用该函数，如下所示：

```
string myString;  
myString = GetString();
```

还有一个函数`GetVal()`，它返回一个`double`值，可在数学表达式中使用它：

```
double myVal;  
double multiplier = 5.3;  
myVal = GetVal() * multiplier;
```

当函数返回一个值时，可以采用以下两种方式修改函数：

- 在函数声明中指定返回值的类型，但不使用关键字`void`。
- 使用`return`关键字结束函数的执行，把返回值传送给调用代码。

从代码角度看，对于我们讨论的控制台应用程序函数，其使用返回值的形式如下所示：

```
static<returnType>  
  
><FunctionName  
  
>()  
{  
    ...  
    return<returnValue>  
  
>;  
}
```

这里唯一的限制是<returnValue>必须是<returnType>类型的值，或者可以隐式转换为该类型。但是，<returnType>可以是任何类型，包括前面介绍的较复杂类型。这段代码可以很简单：

```
static double GetVal()  
{  
    return 3.2;  
}
```

但是，返回值通常是函数执行的一些处理的结果。上面的结果使用

const变量也可以简单地实现。

当执行到return语句时，程序会立即返回调用代码。这条语句后面的代码都不会执行。但这并不意味着return语句只能放在函数体的最后一行。可以在前边的代码里使用return，例如放在分支逻辑之后。把return语句放在for循环、if块或其他结构中会使该结构立即终止，函数也立即终止。例如：

```
static double GetVal()
{
    double checkVal;

    // checkVal assigned a value through some logic (not shown)

    if (checkVal<5)

        return 4.7;

    return 3.2;
```

```
}
```

根据checkVal的值，将返回两个值中的一个。这里的唯一限制是，必须在函数的闭合花括号 } 之前处理return语句。下面的代码是不合法的：

```
static double GetVal()
{
    double checkVal;
    // checkVal assigned a value through some logic.
    if (checkVal<5)
        return 4.7;
}
```

如果checkVal>=5，就不会执行到return语句，这是不允许的。所有处理路径都必须执行到return语句。大多数情况下，编译器会检查是否执行到return语句，如果没有，就给出错误“并不是所有的处理路径都返回一个值”。

执行一行代码的函数可使用C# 6引入的一个功能：表达式体方法（expression-bodied method）。以下函数模式使用=>（Lambda箭头）来实现这一功能。

```
static<returnType>
FunctionName
```

```
>() =><myVal1 * myVal2>;
```

例如，C# 6之前的Multiply()函数如下：

```
static double Multiply(double myVal1, double myVal2)
{
    return myVal1 * myVal2;
}
```

现在可以使用=>（Lambda箭头）编写它。下述代码用更简单和统一的方式表达方法的意图：

```
static double Multiply(double myVal1, double myVal2) => myVal1
```

6.1.2 参数

当函数接受参数时，必须指定以下内容：

- 函数在其定义中指定接受的参数列表，以及这些参数的类型。
- 在每个函数调用中提供匹配的实参列表。

注意：仔细阅读C#规范会发现形参（parameter）和实参（argument）之间存在一些细微的区别：形参是函数定义的一部分，而实参则由调用代码传递给函数。但是，这两个术语通常被简单地称为参数，似乎没有人对此感到十分不满。

示例代码如下所示，其中可以有任意数量的参数，每个参数都有类型和名称：

```
static<returnType

><FunctionName

>(<paramType

><paramName

>, ...)
{
    ...
    return<returnValue

>;
}
```

参数之间用逗号隔开。每个参数都在函数的代码中用作一个变量。例如，下面是一个简单的函数，带有两个double参数，并返回它们的乘积：

```
static double Product(double param1, double param2) => param1
```

下面看一个较复杂的示例：

试一试：通过函数交换数据（1）：**Ch06Ex02\Program.cs**

（1）在C:\BegVCSharp\Chapter06目录中创建一个新的控制台应用程序Ch06Ex02。

（2）把下列代码添加到Program.cs中：

```
class Program
{
    static int MaxValue(int[] intArray)

    {

        int maxVal = intArray[0];

        for (int i = 1; i<intArray.Length; i++)
```

```
{
```

```
    if (intArray[i] > maxVal)
```

```
        maxVal = intArray[i];
```

```
}
```

```
return maxVal;
```

```
}
```

```
static void Main(string[] args)
{
    int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };

    int maxVal = MaxValue(myArray);

    WriteLine($"The maximum value in myArray is {maxVal}");

    ReadKey();
}
}
```

(3) 执行代码，结果如图6-2所示。

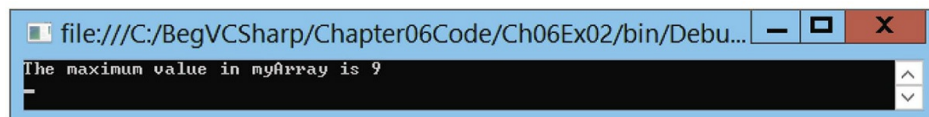


图6-2

示例说明

这段代码包含一个函数，它执行的任务就是本章开头的示例函数所完成的任务。该函数以一个整数数组作为参数，并返回该数组中的最大值。该函数的定义如下所示：

```
static int MaxValue(int[] intArray)
{
    int maxVal = intArray[0];
    for (int i = 1; i<intArray.Length; i++)
    {
        if (intArray[i] > maxVal)
            maxVal = intArray[i];
    }
    return maxVal;
}
```

函数MaxValue()定义了一个参数，即int数组intArray，它还有一个int类型的返回值。最大值的计算是很简单的。局部整型变量maxVal初始化为数组中的第一个值，然后把这个值与数组中后面的每个元素依次进行比较。如果一个元素的值比maxVal大，就用这个值代替当前的maxVal值。循环结束时，maxVal就包含数组中的最大值，用return语句返回。

Main()中的代码声明并初始化一个简单的整数数组，用于MaxValue()函数：

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
```


调用MaxValue(), 把一个值赋给int变量maxVal:

```
int maxVal = MaxValue(myArray);
```

接着, 使用WriteLine()把这个值写到屏幕上:

```
WriteLine($"The maximum value in myArray is {maxVal}");
```

1. 参数匹配

在调用函数时, 必须使提供的参数与函数定义中指定的参数完全匹配, 这意味着要匹配参数的类型、个数和顺序。例如, 下面的函数:

```
static void MyFunction(string myString, double myDouble)
{
    ...
}
```

不能使用下面的代码调用:

```
MyFunction(2.6, "Hello");
```

这里试图把一个double值作为第一个参数传递, 把string值作为第二个参数传递, 参数顺序与函数声明中定义的顺序不匹配。这段代码不能编译, 因为参数类型是错误的。本章后面的“重载函数”一节将介绍解决这个问题一个有效技术。

2. 参数数组

C#允许为函数指定一个（只能指定一个）特殊参数，这个参数必须是函数定义中的最后一个参数，称为参数数组。参数数组允许使用个数不定的参数调用函数，可使用`params`关键字定义它们。

参数数组可以简化代码，因为在调用代码中不必传递数组，而是传递同类型的几个参数，这些参数会放在可在函数中使用的一个数组中。

定义使用参数数组的函数时，需要使用下列代码：

```
static<returnType>
FunctionName(
    <p1Type> p1Name, ...,
    params<type> array
)[]<name>
```

```

>)
{
    ...
    return<returnValue>

>;
}

```

使用下面的代码可以调用该函数：

```

<FunctionName

>( <p1

>, ..., <val1

>, <val2

>, ... )

```

其中<val1 >和<val2 >等都是<type >类型的值，用于初始化<name>数组。可以指定的参数个数几乎不受限制，但它们都必须是<type >类型。甚至根本不必指定参数。

下面的示例定义并使用带有params类型参数的函数。

试一试：通过函数交换数据（2）：**Ch06Ex03\Program.cs**

（1）在C:\BegVCSharp\Chapter06目录中创建一个新的控制台应用程序Ch06Ex03。

（2）把下述代码添加到Program.cs中：

```
class Program
{
    static int SumVals(params int[] vals)

    {

        int sum = 0;

        foreach (int val in vals)
```

```
{
```

```
    sum += val;
```

```
}
```

```
return sum;
```

```
}
```

```
static void Main(string[] args)
```

```
{
```

```
    int sum = SumVals(1, 5, 2, 9, 8);
```

```
        WriteLine($"Summed Values = {sum}");

        ReadKey();

    }
}
```

(3) 执行代码，结果如图6-3所示。

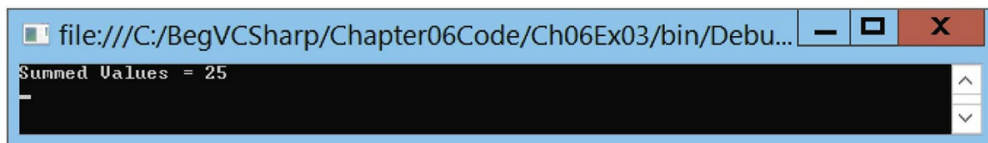


图6-3

示例说明

这个示例用关键字params定义函数sumVals()，该函数可以接受任意个int参数（但不接受其他类型的参数）：

```
static int SumVals(params int[] vals)
{
    ...
}
```

这个函数对vals数组中的值进行迭代，将这些值加在一起，返回其结果。

在Main()中，用5个整型参数调用函数SumVals():

```
int sum = SumVals(1, 5, 2, 9, 8);
```

也可以用0、1、2或100个整型参数调用这个函数——参数的数量不受限制。

注意： C#引入了指定函数参数的新方式，包括用一种可读性更好的方式来包含可选参数。第13章将介绍这些方法，该章讨论C#语言。

3. 引用参数和值参数

本章迄今定义的所有函数都带有值参数。其含义是：在使用参数时，是把一个值传递给函数使用的一个变量。在函数中对此变量的任何修改都不影响函数调用中指定的参数。例如，下面的函数使传递过来的参数值加倍，并显示出来：

```
static void ShowDouble(int val)
{
    val *= 2;
    WriteLine($"val doubled = {0}", val);
}
```

```
}
```

参数val在这个函数中被加倍，如果按以下方式调用它：

```
int myNumber = 5;
WriteLine($"myNumber = {myNumber}");
ShowDouble(myNumber);
WriteLine($"myNumber = {myNumber}", );
```

输出到控制台的文本如下所示：

```
myNumber = 5
val doubled = 10
myNumber = 5
```

把myNumber作为一个参数，调用ShowDouble()并不影响Main()中myNumber的值，即使把myNumber赋值给val后将val加倍，myNumber的值也不变。

这很不错，但如果要改变myNumber的值，就会有问题。可以使用一个为myNumber返回新值的函数：

```
static int DoubleNum(int val)
{
    val *= 2;
    return val;
}
```

并使用下面的代码调用它：


```
int myNumber = 5;
WriteLine($"myNumber = {myNumber}");
myNumber = DoubleNum(myNumber);
```

```
WriteLine($"myNumber = {myNumber}");
```

但这段代码一点也不直观，且不能改变用作参数的多个变量值（因为函数只有一个返回值）。

此时可以通过“引用”传递参数。即函数处理的变量与函数调用中使用的变量相同，而不仅仅是值相同的变量。因此，对这个变量进行的任何改变都会影响用作参数的变量值。为此，只需使用`ref`关键字指定参数：

```
static void ShowDouble(ref int val)

{
    val *= 2;
    WriteLine($"val doubled = {val}");
}
```

在函数调用中再次指定它（这是必需的）：

```
int myNumber = 5;
```

```
WriteLine($"myNumber = {myNumber}",);  
ShowDouble(ref myNumber);
```

```
WriteLine($"myNumber = {myNumber}");
```

输出到控制台的文本如下所示：

```
myNumber = 5  
val doubled = 10  
myNumber = 10
```

用作ref参数的变量有两个限制。首先，函数可能会改变引用参数的值，所以必须在函数调用中使用“非常量”变量。所以，下面的代码是非法的：

```
const int myNumber = 5;
```

```
WriteLine($"myNumber = {myNumber}",);  
ShowDouble(ref myNumber);  
WriteLine($"myNumber = {myNumber}");
```

其次，必须使用初始化过的变量。C#不允许假定ref参数在使用它的函数中初始化，下面的代码也是非法的：

```
int myNumber;
```

```
ShowDouble(ref myNumber);  
WriteLine("myNumber = {myNumber}");
```

4. 输出参数

除了按引用传递值外，还可以使用out关键字，指定所给的参数是一个输出参数。out关键字的使用方式与ref关键字相同（在函数定义和函数调用中用作参数的修饰符）。实际上，它的执行方式与引用参数几乎完全一样，因为在函数执行完毕后，该参数的值将返回给函数调用中使用的变量。但是，二者存在一些重要区别：

- 把未赋值的变量用作ref参数是非法的，但可以把未赋值的变量用作out参数。
- 另外，在函数使用out参数时，必须把它看成尚未赋值。

即调用代码可以把已赋值的变量用作out参数，但存储在该变量中的值会在函数执行时丢失。

例如，考虑前面返回数组中最大值的MaxValue()函数，略微修改该函数，获取数组中最大值的元素索引。为简单起见，如果数组中有多个元素的值都是这个最大值，只提取第一个最大值的索引。为此，修改函数，添加一个out参数，如下所示：

```
static int MaxValue(int[] intArray, out int maxIndex)
```

```
{  
    int maxVal = intArray[0];  
    maxIndex = 0;  
  
    for (int i = 1; i<intArray.Length; i++)  
    {  
        if (intArray[i] > maxVal)  
        {  
  
            maxVal = intArray[i];  
            maxIndex = i;  
  
        }  
  
    }  
  
    return maxVal;  
}
```

可采用以下方式使用该函数：

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };  
int maxIndex;  
WriteLine($"The maximum value in myArray is  
           {MaxValue(myArray, out maxIndex)}");  
WriteLine($"The first occurrence of this value is at element  
           {maxIndex + 1}");
```

结果如下：

```
The maximum value in myArray is 9  
The first occurrence of this value is at element 7
```

注意，必须在函数调用中使用out关键字，就像ref关键字一样。

6.2 变量的作用域

在上一节中，读者可能想知道为什么需要利用函数交换数据。原因是C#中的变量仅能从代码的本地作用域访问。给定的变量有一个作用域，在这个作用域外是不能访问该变量的。

变量的作用域是一个重要主题，最好用一个示例加以说明。下面的示例将演示在一个作用域中定义变量，但试图在另一个作用域中使用该变量的情形。

试一试：变量的作用域：**Ch06Ex01\Program.cs**

(1) 对Ch06Ex01中的Program.cs进行如下修改：

```
class Program
{
    static void Write()
    {
        WriteLine($"myString = {myString}");
    }

    static void Main(string[] args)
```

```
{  
    string myString = "String defined in Main()";  
  
    Write();  
    ReadKey();  
}  
}
```

(2) 编译代码，注意显示在任务列表中的错误和警告：

```
The name 'myString' does not exist in the current context  
The variable 'myString' is assigned but its value is never use
```

示例说明

什么地方出错了？不能在Write()函数中访问在应用程序主体（Main()函数）中定义的变量myString。

原因在于变量是有作用域的，在相应作用域中，变量才是有效的。这个作用域包括定义变量的代码块和直接嵌套在其中的代码块。函数中的代码块与调用它们的代码块是不同的。在Write()中，没有定义myString，在Main()中定义的myString则超出了作用域——它只能在Main()中使用。

实际上，在Write()中可以有一个完全独立的变量myString。修改代码，如下所示：

```

class Program
{
    static void Write()
    {
        string myString = "String defined in Write()";

        WriteLine("Now in Write()");

        WriteLine($"myString = {myString}");
    }
    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
        WriteLine("\nNow in Main()");
        WriteLine($"myString = {myString}");
        ReadKey();
    }
}

```

这段代码就可以编译，输出结果如图6-4所示。

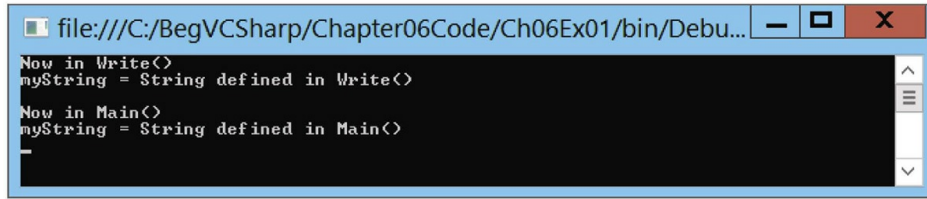


图6-4

这段代码执行的操作如下：

- Main()定义和初始化字符串变量myString。
- Main()把控制权传送给Write()。
- Write()定义和初始化字符串变量myString，它与Main()中定义的myString变量完全不同。
- Write()把一个字符串输出到控制台，该字符串包含在Write()中定义的myString的值。
- Write()把控制权传送回Main()。
- Main()把一个字符串输出到控制台，该字符串包含在Main()中定义的myString的值。

其作用域以这种方式覆盖一个函数的变量称为局部变量。还有一种全局变量，其作用域可覆盖多个函数。修改代码，如下所示：

```
class Program
{
    static string myString;

    static void Write()
```

```

{
    string myString = "String defined in Write()";
    WriteLine("Now in Write()");
    WriteLine($"Local myString = {myString}");

    WriteLine($"Global myString = {Program.myString}");

}

static void Main(string[] args)
{
    string myString = "String defined in Main()";
    Program.myString = "Global string";

    Write();
    WriteLine("\nNow in Main()");
    WriteLine($"Local myString = {myString}");

    WriteLine($"Global myString = {Program.myString}");

```

```
        ReadKey();  
    }  
}
```

执行结果如图6-5所示。



图6-5

这里添加了另一个变量`myString`，这次进一步加深了代码中的名称层次。这个变量定义如下：

```
static string myString;
```

注意，这里也需要`static`关键字。在此类控制台应用程序中，必须使用`static`或`const`关键字来定义这种形式的全局变量。如果要修改全局变量的值，就需要使用`static`，因为`const`禁止修改变量的值。

为区分这个变量和`Main()`与`Write()`中的同名局部变量，必须用一个完整限定的名称为变量名分类，参见第3章。这里把全局变量称为`Program.myString`。注意，只有在全局变量和局部变量同名时，才需要这么做。如果没有局部`myString`变量，就可以使用`myString`表示全局变量，而不需要使用`Program.myString`。如果局部变量和全局变量同名，

会屏蔽全局变量。

全局变量的值在Main()中设置如下：

```
Program.myString = "Global string";
```

全局变量在Write()中可以通过如下语句访问：

```
WriteLine($"Global myString = {Program.myString}");
```

为什么不能使用这个技术通过函数交换数据，而要使用前面介绍的参数来交换数据？有时，这确实是一种交换数据的首选方式，例如编写一个对象，用作插件，或者在较大项目中使用的短脚本。但许多情况下不应使用这种方式。使用全局变量的最常见问题与并发性的管理相关。例如，可以编写一个全局变量来读取一个类的众多方法或读取不同的线程。如果大量的线程和方法可以写入全局变量，能确定全局变量中的值是有效数据吗？没有额外的同步代码，就不能确定。此外，全局变量的真正意图可能被遗忘，以后因为其他原因再次使用它。因此，是否使用全局变量取决于函数的用途。

使用全局变量的问题在于，它们通常不适合于“常规用途”的函数——这些函数能处理我们所提供的任意数据，而不仅限于处理特定全局变量中的数据。详见本章后面的内容。

6.2.1 其他结构中变量的作用域

上一节的一个要点不是只与函数之间的变量作用域有关：变量的作

用域包含定义它们的代码块和直接嵌套在其中的代码块。接下来要讨论的代码可在本章下载文件的VariableScopeInLoops\Program.cs中找到。这一点也适用于其他代码块，例如分支和循环结构的代码块。考虑下面的代码：

```
int i;
for (i = 0; i<10; i++)
{
    string text = "Line " + Convert.ToString(i);
    WriteLine($"{text}");
}
WriteLine($"Last text output in loop: {text}");
```

字符串变量text是for循环的局部变量，这段代码不能编译，因为在该循环外部调用的WriteLine()试图使用该字符串变量，但是在循环外部该字符串变量会超出作用域。修改代码，如下所示：

```
int i;
string text;

for (i = 0; i<10; i++)
{
    text = "Line " + Convert.ToString(i);
```

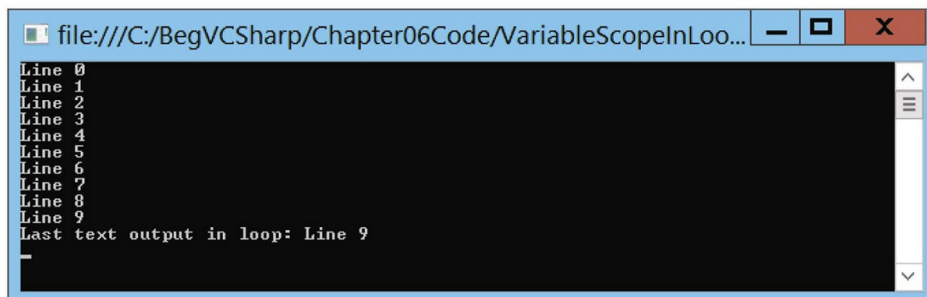
```
        WriteLine($"{text}");
    }
    WriteLine($"Last text output in loop: {text}");
```

这段代码也会失败，原因是必须在使用变量前对其进行声明和初始化，但text只在for循环中初始化。由于没有在循环外进行初始化，赋给text的值在循环块退出时就丢失了。但可以进行如下修改：

```
int i;
string text = "";

for (i = 0; i<10; i++)
{
    text = "Line " + Convert.ToString(i);
    WriteLine($"{text}");
}
WriteLine($"Last text output in loop: {text}");
```

这次text是在循环外部初始化的，可以访问它的值。这段简单代码的执行结果如图6-6所示。



```
file:///C:/BegVCSharp/Chapter06Code/VariableScopeInLoo...
Line 0
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Last text output in loop: Line 9
```

图6-6

在循环中最后赋给text的值可以在循环外部访问。可以看出，这个主题的内容需要花一点时间来掌握。在前面的示例中，循环之前将空字符串赋给text，而在循环之后的代码中，text就不会是空字符串了，其原因可能一下子看不出来。

这种情况的解释涉及分配给text变量的内存空间，实际上任何变量都是这样。只声明一个简单变量类型，并不会引起其他变化。只有在给变量赋值后，这个值才会被分配一块内存空间。如果这种分配内存空间的行为在循环中发生，该值实际上定义为一个局部值，在循环外部会超出其作用域。

即使变量本身未局部化到循环上，其包含的值却会局部化到该循环上。但在循环外部赋值可以确保该值是主体代码的局部值，在循环内部它仍处于其作用域中。这意味着变量在退出主体代码块之前是没有超出作用域的，所以可在循环外部访问它的值。

幸好，C#编译器可检测变量作用域的问题，根据它生成的错误信息修正程序有助于我们理解变量的作用域问题。

6.2.2 参数和返回值与全局数据

本节将详细介绍如何通过全局数据以及参数和返回值与函数交换数据。首先分析下面的代码：

```
class Program
```

```
{  
    static void ShowDouble(ref int val)
```

```
{
```

```
    val *= 2;
```

```
    WriteLine($"val doubled = {val}");
```

```
}
```

```
static void Main(string[] args)
```

```
{
```

```
    int val = 5;
```



```
WriteLine($"val = {val}");
```

```
ShowDouble(ref val);
```

```
WriteLine($"val = {val}");
```

```
}
```

```
}
```

注意：这段代码与本章前面的代码稍有不同，在前面的示例中，在Main()中使用了变量名myNumber，这说明局部变量可以具有相同的名称，且不会相互干涉。

和下面的代码比较：

```
class Program
{
    static int val;
```

```
static void ShowDouble()
```

```
{
```

```
    val *= 2;
```

```
    WriteLine($"val doubled = {val}");
```

```
}
```

```
static void Main(string[] args)
```

```
{
```

```
    val = 5;
```

```
    WriteLine($"val = {val}");
```

```
    ShowDouble();
```

```
        WriteLine($"val = {val}");  
    }  
}
```

这两个ShowDouble()函数的结果是相同的。

使用哪种方法并没有什么硬性规定，这两种方法都十分有效，但需要考虑一些规则。

首先，在第一次讨论这个问题时就提到过，使用全局值的ShowDouble()版本只使用全局变量val。为使用这个版本，必须使用这个全局变量。这会对该函数的灵活性有轻微的限制，如果要存储结果，就必须总是把这个全局变量值复制到其他变量中。另外，全局数据可能在应用程序的其他地方被代码修改，这会导致预料不到的结果（其值可能会改变，等我们认识到这一点时为时已晚）。

当然，也可以说，这种简化实际上使代码更难理解。显式指定参数可以一眼看出发生了什么改变。例如对于FunctionName（val1, out val2）函数调用，马上就可以知道val1和val2都是要考虑的重要变量，在函数执行完毕后，会为val2赋予一个新值。反之，如果这个函数不带参数，就不能对它处理了什么数据做任何假设。

总之，可以自由选择使用哪种技术来交换数据。一般情况下，最好使用参数，而不使用全局数据，但有时使用全局数据更合适，使用这种技术并没有错。

6.3 Main()函数

前面介绍了创建和使用函数时涉及的大多数简单技术，下面详细论述Main()函数。

Main()是C#应用程序的入口点，执行这个函数就是执行应用程序。也就是说，在执行过程开始时，会执行Main()函数，在Main()函数执行完毕时，执行过程就结束了。

这个函数可以返回void或int，有一个可选参数string[] args。Main()函数可使用如下4种版本：

```
static void Main()  
static void Main(string[] args)  
static int Main()  
static int Main(string[] args)
```

上面的第3和第4个版本返回一个int值，它们可以用于表示应用程序的终止方式，通常用作一种错误提示（但这不是强制的）。一般情况下，返回0反映了“正常”的终止（即应用程序已经执行完毕，并安全地终止）。

Main()的可选参数args是从应用程序的外部接受信息的方法，这些信息在运行应用程序时以命令行参数的形式指定。

在执行控制台应用程序时，指定的任何命令行参数都放在这个args数组中，接着可以根据需要在应用程序中使用这些参数。下面用一个示

例来说明。这个示例可以指定任意数量的命令行参数，每个参数都被输出到控制台。

试一试：命令行参数：**Ch06Ex04\Program.cs**

(1) 在C:\BegVCSharp\Chapter06目录中创建一个新的控制台应用程序Ch06Ex04。

(2) 把下列代码添加到Program.cs中：

```
class Program
{
    static void Main(string[] args)
    {
        WriteLine($"{args.Length} command line arguments were sp

        foreach (string arg in args)

            WriteLine(arg);
```

```
ReadKey();
```

```
}
```

```
}
```

(3) 打开项目的属性页面（在Solution Explorer窗口中右击Ch06Ex04项目名称，然后选择Properties选项）。

(4) 选择Debug页面，在Command line arguments设置中添加所希望的命令行参数，如图6-7所示。

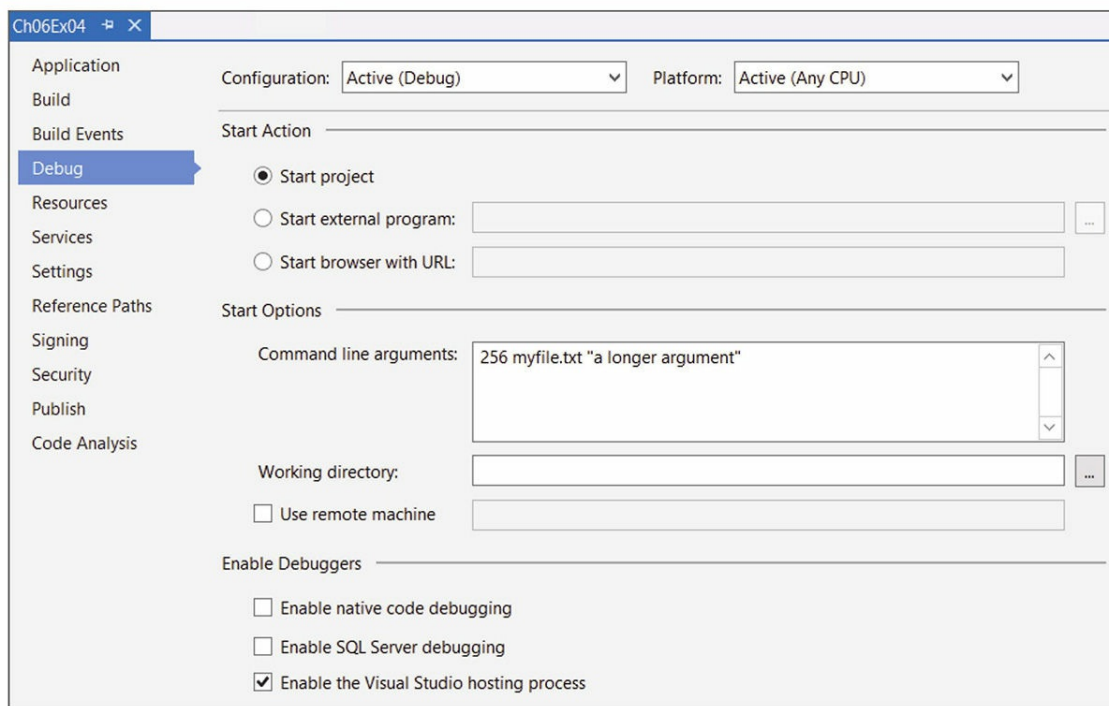


图6-7

(5) 运行应用程序，输出结果如图6-8所示。

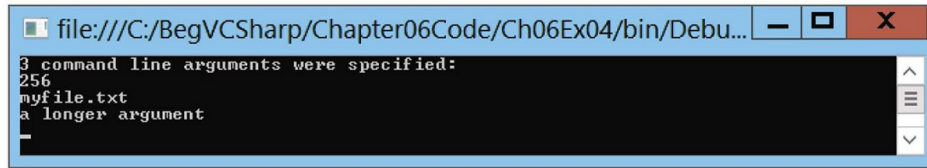


图6-8

示例说明

这里使用的代码非常简单：

```
WriteLine($"{args.Length} command line arguments were specifie  
foreach (string arg in args)  
    WriteLine(arg);
```

使用args参数与使用其他字符串数组类似。我们没有对参数进行任何异样的操作，只是把指定信息写到屏幕上。在本例中，通过IDE中的项目属性提供参数，这是一种便捷方式，只要在IDE中运行应用程序，就可以使用相同的命令行参数，不必每次都在命令行提示窗口中键入它们。在项目输出所在的目录

（C:\BegCSharp\Chapter06\Ch06Ex04\Ch06Ex04\bin\Debug）下打开命令提示窗口，键入下述代码，也可以得到同样的结果：

```
Ch06Ex04 256 myFile.txt "a longer argument"
```

每个参数都用空格分开。如果参数包含空格，就可以用双引号把参数括起来，这样才不会把这个参数解释为多个参数。

6.4 结构函数

第5章介绍了结构类型，它可在一个地方存储多个数据元素，但实际上结构可以做的工作远不止这一点。例如，除了数据，结构还可以包含函数。这初看起来很奇怪，但实际上是非常有用的。例如，考虑以下结构：

```
struct CustomerName
{
    public string firstName, lastName;
}
```

如果变量类型是`CustomerName`，并且要在控制台上输出一个完整的姓名，就必须使用姓、名构成该姓名。例如，对于`CustomerName`变量`myCustomer`，可以使用下述语法：

```
CustomerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
WriteLine($"{myCustomer.firstName} {myCustomer.lastName}");
```

把函数添加到结构中，就可以集中处理常见任务，从而简化这个过程。可以把合适的函数添加到结构类型中，如下所示：

```
struct CustomerName
{
```



```
public string firstName, lastName;  
  
public string Name() => firstName + " " + lastName;  
  
}
```

看起来这与本章前面的其他函数类似，只不过没有使用static修饰符。本书后面将阐明其原因，现在知道该关键字不是结构函数所必需的即可。这个函数的用法如下所示：

```
CustomerName myCustomer;  
myCustomer.firstName = "John";  
myCustomer.lastName = "Franklin";  
WriteLine(myCustomer.Name());
```

这个语法比前面的语法简单得多，也更容易理解。注意，Name()函数可以直接访问firstName和lastName结构成员。在customerName结构中，它们可以被看成全局成员。

6.5 函数的重载

本章前面提到过，在调用函数时，必须匹配函数的签名。这表明，需要有不同的函数操作不同类型的变量。函数重载允许创建多个同名函数，每个函数可使用不同的参数类型。例如，前面使用了下述代码，其中包含函数MaxValue()：

```
class Program
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i<intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }
    static void Main(string[] args)
    {
        int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
        int maxVal = MaxValue(myArray);
        WriteLine("The maximum value in myArray is {maxVal}");
        ReadKey();
    }
}
```

```
}  
}
```

这个函数只能用于处理int数组。可为不同的参数类型提供不同名称的函数，例如把上述函数重命名为IntArrayMaxValue()，添加诸如DoubleArrayMaxValue()的函数来处理其他类型。还有一种方法，即在代码中添加如下函数：

```
static double MaxValue(double[] doubleArray)  
{  
    double maxVal = doubleArray[0];  
    for (int i = 1; i<doubleArray.Length; i++)  
    {  
        if (doubleArray[i] > maxVal)  
            maxVal = doubleArray[i];  
    }  
    return maxVal;  
}
```

这里的区别是使用了double值。函数名称MaxValue()是相同的，但其签名是不同的。这是因为如前所述，函数的签名包含函数的名称及其参数。用相同签名来定义两个函数是错误的，但因为这里的两个函数的签名不同，所以没有问题。

注意： 函数的返回类型不是其签名的一部分，所以不能定义两个仅返回类型不同的函数，它们实际上有相同的签名。

添加了前面的代码后，现在有两个版本的MaxValue()，它们的参数是int和double数组，分别返回int或double类型的最大值。

这种代码的优点是不必显式地指定要使用哪个函数。只需提供一个数组参数，就可以根据使用的参数类型执行相应的函数。

此时，应注意VS中IntelliSense的另一项功能。如果在应用程序中有上述两个函数，而且要在Main()或其他函数中键入函数的名称，IDE就可以显示出可用的重载函数。如果键入下面的代码：

```
double result = MaxValue(
```

IDE就会提供两个MaxValue()版本的信息，可使用上下箭头键在其间滚动，如图6-9所示。

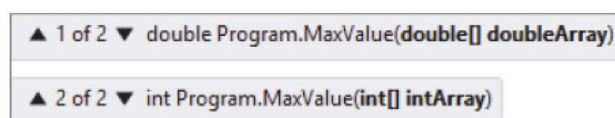


图6-9

在重载函数时，应包括函数签名的所有方面。例如，有两个不同函数，它们分别带有值参数和引用参数：

```
static void ShowDouble(ref int val)
```

```
{  
    ...  
}
```

```
static void ShowDouble(int val)
```

$$\{ \dots \}$$

}

选用哪个版本完全根据函数调用是否包含`ref`关键字来确定。下面的代码将调用引用版本：

```
ShowDouble(ref val);
```

下面的代码调用值版本:

```
ShowDouble(val);
```

此外，还可以根据参数的个数等来区分函数。

6.6 委托

委托（`delegate`）是一种存储函数引用的类型。这听起来相当深奥，但其机制是非常简单的。委托最重要的用途在本书后面介绍到事件和事件处理时才能解释清楚，但这里也将介绍有关委托的许多内容。委托的声明非常类似于函数，但不带函数体，且要使用`delegate`关键字。委托的声明指定了一个返回类型和一个参数列表。

定义了委托后，就可以声明该委托类型的变量。接着把这个变量初始化为与委托具有相同返回类型和参数列表的函数引用。之后，就可以使用委托变量调用这个函数，就像该变量是一个函数一样。

有了引用函数的变量后，就可以执行无法用其他方式完成的操作。例如，可以把委托变量作为参数传递给一个函数，这样，该函数就可以使用委托调用它引用的任何函数，而且在运行之前不必知道调用的是哪个函数。下面的示例使用委托访问两个函数中的一个。

试一试：使用委托来调用函数：**Ch06Ex05\Program.cs**

（1）在C:\BegVCSharp\Chapter06目录中创建一个新的控制台应用程序Ch06Ex05。

（2）把下列代码添加到Program.cs中：

```
class Program
{
    delegate double ProcessDelegate(double param1, double para

    static double Multiply(double param1, double param2) => pa

    static double Divide(double param1, double param2) => para

    static void Main(string[] args)
    {
        ProcessDelegate process;

        WriteLine("Enter 2 numbers separated with a comma:");

        string input = ReadLine();
```

```
int commaPos = input.IndexOf(',');
```

```
double param1 = ToDouble(input.Substring(0, commaPos));
```

```
double param2 = ToDouble(input.Substring(commaPos + 1,
```

```
input.Length - commaPos - 1));
```

```
WriteLine("Enter M to multiply or D to divide:");
```

```
input = ReadLine();
```

```
if (input == "M")
```



```
        process = new ProcessDelegate(Multiply);

    else

        process = new ProcessDelegate(Divide);

    WriteLine($"Result: {process(param1, param2)}");

    ReadKey();

}
}
```

(3) 执行代码，在看到提示时输入值，结果如图6-10所示。

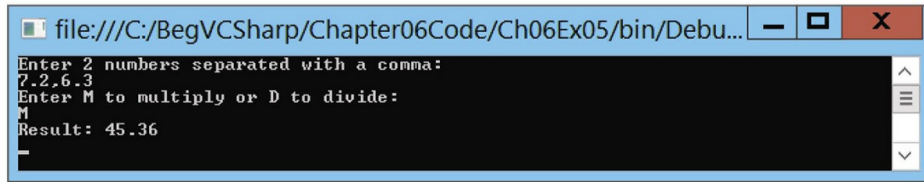


图6-10

示例说明

这段代码定义了一个委托ProcessDelegate，其返回类型和参数与函数Multiply()和Divide()相匹配。注意Multiply()和Divide()方法使用了C# 6引入的=>（Lambda箭头）。

```
static double Multiply(double param1, double param2) => param1
```

委托的定义如下所示：

```
delegate double ProcessDelegate(double param1, double param2);
```

delegate关键字指定该定义是用于委托的，而不是用于函数的（该定义所在的位置与函数定义相同）。接着，该定义指定double返回类型和两个double参数。实际使用的名称可以是任意的，所以可以给委托类型和参数指定任意名称。这里委托名是ProcessDelegate，double参数名是param1和param2。

Main()中的代码首先使用新的委托类型声明一个变量：

```
static void Main(string[] args)
{
    ProcessDelegate process;
```

接着用一些比较标准的C#代码请求由逗号分隔的两个数字，并将这些数字放在两个double变量中：

```
WriteLine("Enter 2 numbers separated with a comma:");
string input = ReadLine();
int commaPos = input.IndexOf(',');
double param1 = ToDouble(input.Substring(0, commaPos));
double param2 = ToDouble(input.Substring(commaPos + 1,
                                         input.Length - commaPos - 1));
```

注意：为说明问题，这里没有验证用户输入的有效性。如果这些是“现实中的”代码，就应花费更多时间来确保在局部变量param1和param2中得到有效的值。

接着询问用户，这两个数字是要相乘还是相除：

```
WriteLine("Enter M to multiply or D to divide:");
input = ReadLine();
```

根据用户的选择，初始化process委托变量：

```
if (input == "M")
    process = new ProcessDelegate(Multiply);
else
    process = new ProcessDelegate(Divide);
```

要把一个函数引用赋给委托变量，需要使用略显古怪的语法。这个

过程比较类似于给数组赋值，必须使用**new**关键字创建一个新委托。在这个关键字的后面，指定委托类型，提供一个引用所需函数的参数，这里也就是**Multiply()**或**Divide()**函数。注意这个参数与委托类型或目标函数的参数不匹配，这是委托赋值的一种独特语法，参数是要使用的函数名且不带括号。

实际上，这里可以使用略微简单的语法：

```
if (input == "M")  
    process = Multiply;  
  
else  
    process = Divide;
```

编译器会发现，**process**变量的委托类型匹配两个函数的签名，于是自动初始化一个委托。可以自行确定使用哪种语法，但一些人喜欢使用较长的版本，因为它更容易一眼看出会发生什么。

最后，使用该委托调用所选的函数。无论委托引用的是什么函数，该语法都是有效的：

```
WriteLine($"Result: {process(param1, param2)}");  
ReadKey();
```

```
}
```

这里把委托变量看成一个函数名。但与函数不同，我们还可以对这个变量执行更多操作，例如，通过参数将其传递给一个函数，如下例所示：

```
static void ExecuteFunction(ProcessDelegate process)
    => process(2.2, 3.3);
```

就像选择一个要使用的“插件”一样，通过把函数委托传递给函数，就可以控制函数的执行。例如，一个函数要对字符串数组按照字母进行排序。对列表排序有几个不同的方法，它们的性能取决于要排序的列表特性。使用委托可以把一个排序算法函数委托传递给排序函数，指定要使用的函数。

委托有许多用途，但如前所述，它们的大多数常见用途主要与事件处理有关，具体内容详见第13章。

6.7 练习

(1) 下面两个函数都存在错误，请指出这些错误。

```
static bool Write()
{
    WriteLine("Text output from function.");
}
static void MyFunction(string label, params int[] args, bool s
{
    if (showLabel)
        WriteLine(label);
    foreach (int i in args)
        WriteLine("{0}", i);
}
```

(2) 编写一个应用程序，该程序使用两个命令行参数，分别把值放在一个字符串和一个整型变量中，然后显示这些值。

(3) 创建一个委托，在请求用户输入时，使用它模拟 Console.ReadLine() 函数。

(4) 修改下面的结构，使其包含一个返回订单总价的函数。

```
struct order
{
```

```

    public string  itemName;
    public int     unitCount;
    public double  unitCost;
}

```

（5）在order结构中添加另一个函数，使其返回如下所示的一个格式化字符串（一行文本，以合适的值替换用尖括号括起来的斜体条目）。

```

Order Information:<unit count

><item name

> items at $<unit cost

> each,
    total cost $<total cost

>

```

附录A给出了练习答案。

6.8 本章要点

主题	要点
定义函数	用函数名、0个或多个参数及返回类型来定义函数。函数的名称和参数统称为函数的签名。可以定义名称相同但签名不同的多个函数——这称为函数重载。也可以在结构类型中定义函数
返回值和参数	函数的返回类型可以是任意类型，如果函数没有返回值，其返回类型就是void。参数也可以是任意类型，由一个用逗号分隔的类型和名称对组成。个数不定的特定类型的参数可以通过参数数组来指定。参数可以指定为ref或out，以便给调用者返回值。调用函数时，所指定的参数的类型和顺序必须匹配函数的定义，并且如果参数定义中使用了ref或out关键字，那么在调用函数时也必须包括对应的ref或out关键字
变量作用域	变量根据定义它们的代码块来界定其使用范围。代码块包括方法和其他结构，例如循环体。可在不同的作用域中定义多个不同的同名变量
命令行参数	在执行应用程序时，控制台应用程序中的Main()函数可以接收传送给应用程序的命令行参数。这些参数用空格隔开，较长的参数可以放在引号中传送
委托	除了直接调用函数外，还可以通过委托调用它们。委托是用返回类型和参数列表定义的变量。给定的委托类型可以匹配返回类型和参数与委托定义相同的方法

第7章 调试和错误处理

本章内容：

- IDE中的调试方法
- C#中的错误处理技术

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 7 Code后，可以找到与本章示例对应的单独文件。

本书到目前为止介绍了在C#中进行简单编程的所有基础知识。本书下一部分将讨论面向对象编程，在此之前先看看C#代码中的调试和错误处理。

代码中有时难免存在错误。无论程序员多么优秀，程序总是会出现一些问题，优秀的程序员必须意识到这一点，并准备好解决这些问题。当然，一些问题比较小，不会影响应用程序的执行，例如，按钮上的拼写错误等，但一些错误可能比较严重，会导致应用程序完全失败（通常称为致命错误），致命错误包括妨碍代码编译的简单错误（语法错误），或者只在运行期间发生的更严重错误。一些错误较难注意到。例

如，也许因为缺少请求的字段，应用程序不能给数据库添加一条记录，或者在其他有限制的环境中把错误数据添加到记录中。应用程序的逻辑在某些方面有瑕疵时，就会产生这样的错误，此类错误称为语义错误（或逻辑错误）。

通常，当应用程序的用户抱怨程序不能正常工作时，开发人员才会知道存在这样的错误。此时需要跟踪代码，确定发生了什么问题，并修改代码，使其按照希望的那样工作。此类情况下，VS的调试功能就可以大显身手了。本章的第一部分就介绍一些调试技巧，并用它们来解决一些常见问题。

此后讨论C#中的错误处理技术。利用它们，可以对可能发生错误的地方采取预防措施，并编写弹性代码来处理可能致命的错误。这些技术是C#语言的一部分，而不是调试功能，但IDE也提供了一些工具来帮助我们处理错误。

7.1 Visual Studio中的调试

前面提到，可以采用两种方式执行应用程序：调试模式或非调试模式。在VS中执行应用程序时，默认在调试模式下执行。例如，按下F5键或单击工具栏中的绿色Start按钮时，就是在调试模式下执行应用程序。要在非调试模式下执行应用程序，应选择Debug|Start Without Debugging，或按下Ctrl+F5键。

VS允许在两种配置下生成应用程序：调试（默认）和发布。使用标准工具栏中的Solution Configurations下拉框可在这两种配置之间切换。

在调试配置下生成应用程序，并在调试模式下运行程序时，并不仅是运行编写好的代码。调试程序包含应用程序的符号信息，所以IDE知道执行每行代码时发生了什么。符号信息意味着跟踪（例如）未编译代码中使用的变量名，这样它们就可以匹配已编译的机器码应用程序中现有的值，而机器码程序不包含便于人们阅读的信息。此类信息包含在.pdb文件中，这些文件位于计算机的Debug目录下。

发布配置会优化应用程序代码，所以我们不能执行以上这些操作。但发布版本运行速度较快。完成了应用程序的开发时，一般应给用户提提供发布版本，因为发布版本不需要调试版本所包含的符号信息。

本节介绍调试技巧，以及如何使用它们找出并修改未按预期方式执行的那些代码，这个过程称为调试。按照这些技术的使用方法把它们分为两部分。一般情况下，可以首先中断程序的执行，再进行调试，或者

注上标记，以便以后加以分析。在VS术语中，应用程序可以处于运行状态，也可以处于中断模式，即暂停正常的执行。下面首先介绍非中断模式（运行期间或正常执行）技术。

7.1.1 非中断（正常）模式下的调试

本书经常使用的一个命令是WriteLine()函数，它可以把文本输出到控制台。在开发应用程序时，这个函数可以方便地获得操作的额外反馈，例如：

```
WriteLine("MyFunc() Function about to be called.");  
MyFunc("Do something.");  
WriteLine("MyFunc() Function execution completed.");
```

这段代码说明了如何获取MyFunc()函数的额外信息。这么做完全正确，但控制台的输出结果会比较混乱。在开发其他类型的应用程序时，如桌面应用程序，没有用于输出信息的控制台。作为一种替代方法，可将文本输出到另一个位置——IDE中的Output窗口。

第2章简要介绍了Error List窗口，提到其他窗口也可以显示在这个位置。其中一个窗口就是Output窗口，在调试时这个窗口非常有用。要显示这个窗口，可以选择View|Output。在这个窗口中，可以查看与代码的编译和执行相关的信息，包括在编译过程中遇到的错误等，还可以将自定义的诊断信息直接写到这个窗口中。该窗口如图7-1所示。

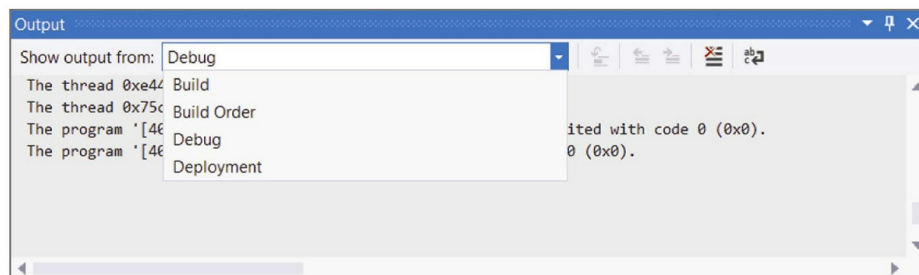


图7-1

注意： 使用Output窗口的下拉菜单可以选择几种模式： Build、Deployment和Debug。 Build和Debug模式分别显示编译和运行期间的信息。本节提到“写入Output窗口”时，实际上是指“写入Output窗口的Debug模式视图”。

另外，还可以创建一个日志文件，在运行应用程序时，会把信息添加到该日志文件中。把信息写入日志文件所用的技巧与把文本写到Output窗口中所用的技巧相同，但需要理解如何从C#应用程序中访问文件系统。我们把这个功能放在后面的章节中加以讨论，因为目前不必了解文件访问技巧也可以完成很多工作。

1. 输出调试信息

在运行期间把文本写入Output窗口是非常简单的。只要用需要的调用替代WriteLine()调用，就可以把文本写到希望的地方。此时可以使用如下两个命令：

- Debug.WriteLine()

- `Trace.WriteLine()`

这两个命令函数的用法几乎完全相同，但有一个重要区别：第一个命令仅在调试模式下运行，而第二个命令还可用于发布程序。实际上，`Debug.WriteLine()`命令甚至不能编译到可发布的程序中，在发布版本中，该命令会消失，这肯定有其优点（编译好的代码文件比较小）。

注意：`Debug.WriteLine()`和`Trace.WriteLine()`方法包含在`System.Diagnostics`名称空间内。`using static`指令只能用于静态类，例如包括`WriteLine()`方法的`System.Console`。

这两个函数的用法与`Console.WriteLine()`是不同的。其唯一的字符串参数用于输出消息，而不需要使用`{X}`语法插入变量值。这意味着必须使用`+`串联运算符等方式在字符串中插入变量值。它们还可以有第二个字符串参数（可选），用于显示输出文本的类别。这样，如果应用程序的不同地方输出了类似的消息，我们马上可以确定Output窗口中显示的是哪些输出信息。

这些函数的一般输出如下所示：

`<category`

`>:<message`

>

例如，下面的语句把MyFunc作为可选的类别参数：

```
Debug.WriteLine("Added 1 to i", "MyFunc");
```

其结果为：

```
MyFunc: Added 1 to i
```

下面的示例按这种方式输出调试信息。

试一试：把文本输出到**Output**窗口：**Ch07Ex01\Program.cs**

（1）在C:\BegVCSharp\Chapter07目录中创建一个新的控制台应用程序Ch07Ex01。

（2）修改代码，如下所示：

```
using System;  
using System.Collections.Generic;  
using System.Diagnostics;
```

```
using System.Linq;  
using System.Text;
```

```
using System.Threading.Tasks;
using static System.Console;
namespace Ch07Ex01
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};

            int[] maxValIndices;

            int maxVal = Maxima(testArray, out maxValIndices);

            WriteLine($"Maximum value {maxVal} found at element ind

            foreach (int index in maxValIndices)
```



```
{
```

```
    WriteLine(index);
```

```
}
```

```
    ReadKey();
```

```
}
```

```
static int Maxima(int[] integers, out int[] indices)
```

```
{
```

```
    Debug.WriteLine("Maximum value search started.");
```

```
indices = new int[1];
```

```
int maxVal = integers[0];
```

```
indices[0] = 0;
```

```
int count = 1;
```

```
Debug.WriteLine(string.Format(
```

```
    $"Maximum value initialized to {maxVal}, at element i
```

```
for (int i = 1; i<integers.Length; i++)
```

```
{
```

```
    Debug.WriteLine(string.Format(
```

```
        $"Now looking at element at index {i}.");
```

```
    if (integers[i] > maxVal)
```

```
    {
```

```
        maxVal = integers[i];
```

```
count = 1;
```

```
indices = new int[1];
```

```
indices[0] = i;
```

```
Debug.WriteLine(string.Format(
```

```
    $"New maximum found. New value is {maxVal}, at
```

```
    element index {i}.");
```

```
}
```

else

{

if (integers[i] == maxVal)

{

count++;

int[] oldIndices = indices;

indices = new int[count];

```
oldIndices.CopyTo(indices, 0);
```

```
indices[count - 1] = i;
```

```
Debug.WriteLine(string.Format(
```

```
    $"Duplicate maximum found at element index {i}.
```

```
    }
```

```
}
```

```
}
```

```
Trace.WriteLine(string.Format(
```

```
$"Maximum value {maxVal} found, with {count} occurrence
```

```
Debug.WriteLine("Maximum value search completed.");
```

```
return maxVal;
```

```
}
```

```
}
```

```
}
```

(3) 在Debug模式下执行代码，结果如图7-2所示。

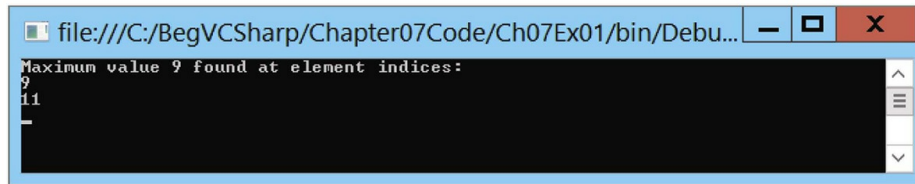


图7-2

(4) 中断应用程序的执行，查看Output窗口中的内容（在Debug模式下），如下所示（有删减）：

...

Maximum value search started.

Maximum value initialized to 4, at element index 0.

Now looking at element at index 1.

New maximum found. New value is 7, at element index 1.

Now looking at element at index 2.

Now looking at element at index 3.

Now looking at element at index 4.

Duplicate maximum found at element index 4.

Now looking at element at index 5.

Now looking at element at index 6.

Duplicate maximum found at element index 6.

Now looking at element at index 7.

New maximum found. New value is 8, at element index 7.

Now looking at element at index 8.

Now looking at element at index 9.

New maximum found. New value is 9, at element index 9.


```
Now looking at element at index 10.  
Now looking at element at index 11.  
Duplicate maximum found at element index 11.  
Maximum value 9 found, with 2 occurrences.  
Maximum value search completed.  
The thread ##### has exited with code 0 (0x0).
```

(5) 使用标准工具栏上的下拉菜单，切换到**Release**模式，如图7-3所示。

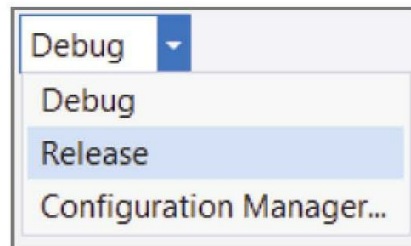


图7-3

(6) 再次运行程序，这次在**Release**模式下运行，并在执行终止时再查看一下Output窗口，结果如下所示（有删减）：

```
...  
Maximum value 9 found, with 2 occurrences.  
The thread ##### has exited with code 0 (0x0).
```

示例说明

这个应用程序是第6章中一个示例的扩展版本，它使用一个函数计算整数数组中的最大值。这个版本也返回一个索引数组，表示最大值在数组中的位置，以便调用代码处理这些元素。

首先在代码开头使用了一个额外的using指令：

```
using System.Diagnostics;
```

这简化了前面讨论的对函数的访问，因为它们包含在System.Diagnostics名称空间中，没有这个using语句，下面的代码：

```
Debug.WriteLine("Bananas");
```

就需要进一步限定，重新编写这行语句，如下所示：

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

Main()中的代码仅初始化一个测试用的整数数组testArray，并声明了另一个整数数组maxValIndices，以存储Maxima()（执行计算的函数）的索引输出结果，接着调用这个函数。函数返回后，代码就会输出结果。

Maxima()稍复杂一些，但用到的代码大部分在前面已经看到过。在数组中进行搜索的方式与第6章的MaxVal()函数类似，但要用一条记录来存储最大值的索引。

特别需要注意用来跟踪索引的函数（而不是输出调试信息的那些代码行）。Maxima()并没有返回一个足以存储源数组中每个索引的数组（需要与源数组有相同的维数），而是返回一个正好能容纳搜索到的索引的数组。这可通过在搜索过程中连续重建不同长度的数组来实现。必须这么做，因为一旦创建好数组，就不能重新设置长度。

开始搜索时，假定源数组（integers）中的第一个元素就是最大值，而且数组中只有一个最大值。因此可以为maxVal（函数的返回值，即搜

索到的最大值)和indices (out参数数组, 存储搜索到的最大值的索引) 设置值。maxVal被赋予integers中第一个元素的值, indices被赋予一个值0, 即数组中第一个元素的索引。在变量count中存储搜索到的最大值的个数, 以便跟踪indices数组。

函数的主体是一个循环, 它迭代integers数组中的各个值, 但忽略第一个值, 因为它已经处理过这个值。每个值都与maxVal的当前值进行比较, 如果maxVal更大, 就忽略该值。如果当前处理的值比maxVal大, 就修改maxVal和indices, 以反映这种情况。如果当前处理的值与maxVal相等, 就递增count, 用一个新数组替代indices。这个新数组比旧indices数组多一个元素, 包含搜索到的新索引。

最后一个功能的代码如下所示:

```
if (integers[i] == maxVal)
{
    count++;
    int[] oldIndices = indices;
    indices = new int[count];
    oldIndices.CopyTo(indices, 0);
    indices[count - 1] = i;
    Debug.WriteLine(string.Format(
        $"Duplicate maximum found at element index {i}."));
}
```

这段代码把旧indices数组备份到if代码块的oldIndices局部整型数组中。注意使用<array> . CopyTo()函数把oldIndices中的值复制到新的indices数组中。这个函数的参数是一个目标数组和一个用于复制第一个

元素的索引，并把所有的值都粘贴到目标数组中。

在代码中，各个文本部分都使用`Debug.WriteLine()`和`Trace.WriteLine()`函数进行输出，这些函数使用`string.Format()`函数把变量值嵌套在字符串中，其方式与`WriteLine()`相同。这比使用+串联运算符更高效。

在Debug模式下运行应用程序时，其最终结果是一条完整记录，它记述了在循环中计算出结果所采取的步骤。在Release模式下，仅能看到计算的最终结果，因为没有调用`Debug.WriteLine()`函数。

2. 跟踪点

另一种把信息输出到Output窗口的方法是使用跟踪点（`tracepoint`）。这是VS的一个功能，而不是C#的功能，但其作用与使用`Debug.WriteLine()`相同。它实际上是输出调试信息且不修改代码的一种方式。

为了演示跟踪点，可用它们替代上一个示例中的调试命令（请参阅本章的下载代码中的`Ch07Ex01TracePoints`文件）。添加跟踪点的过程如下：

（1）把光标放在要插入跟踪点的代码行上。跟踪点会在执行这行代码之前被处理。

（2）右击该行代码，选择`Breakpoint|Insert Tracepoint`。右击代码行旁边的红圆，选择`Settings`菜单项。

（3）选中Actions复选框，在Log a message部分的Message文本框中键入要输出的字符串。如果要输出变量值，应把变量名放在花括号中。

（4）单击OK按钮。在包含跟踪点的代码行的左边会出现一个红色菱形，该行代码也会突出显示为红色。

看一下添加跟踪点的对话框标题和所需要的菜单选项，显然，跟踪点是断点的一种形式（可以暂停应用程序的执行，就像断点一样）。断点一般用于更高级的调试目的，本章稍后将介绍断点。

图7-4显示了Ch07Ex01TracePoints中第32行所需的跟踪点。在删除已有的Debug.WriteLine()语句后，对代码行编号。

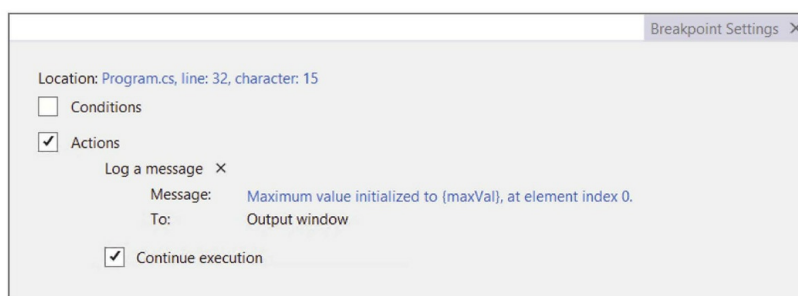


图7-4

还有一个窗口可用于快速查看应用程序中的跟踪点。要显示这个窗口，可从VS菜单中选择Debug|Windows|Breakpoints。这是显示断点的通用窗口（如前所述，跟踪点是断点的一种形式）。可以定制显示的内容，从这个窗口的Columns下拉框中添加When Hit列，显示与跟踪点关系更密切的信息。图7-5显示的窗口配置了该列，还显示了添加到Ch07Ex01TracePoints中的所有跟踪点。

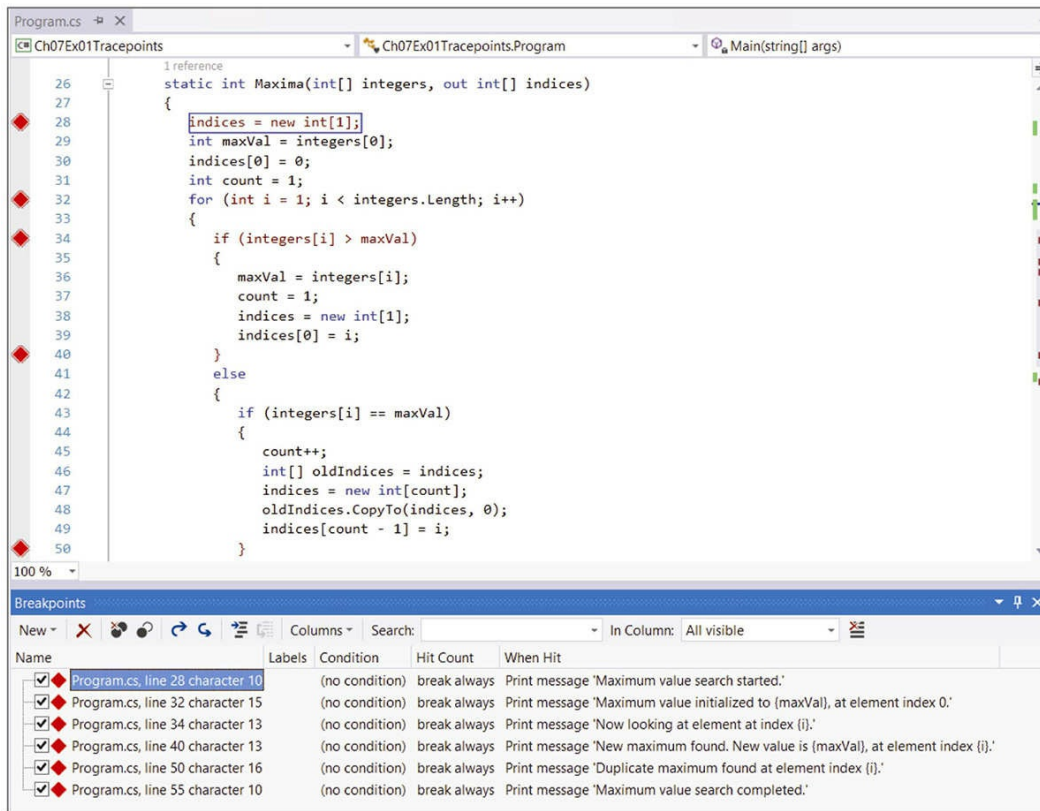


图7-5

在调试模式下执行这个应用程序，会得到与前面完全相同的结果。在代码窗口中右击跟踪点，或者利用Breakpoints窗口，可以删除或临时禁用跟踪点。在Breakpoints窗口中，跟踪点左边的复选框指示是否启用跟踪点；禁用的跟踪点未被选中，在代码窗口中显示为菱形框，而不是实心菱形。

3. 诊断输出与跟踪点

前面介绍了两种输出相同信息的方法，下面分析它们的优缺点。首先，跟踪点与Trace命令并不等价，也就是说，不能使用跟踪点在发布版本中输出信息。这是因为跟踪点并没有包含在应用程序中。跟踪点由

VS处理，在应用程序的已编译版本中，跟踪点是不存在的。只有应用程序运行在VS调试器中时，跟踪点才起作用。

跟踪点的主要缺点也是其主要优点，即它们存储在VS中，因此可以在需要时便捷地添加到应用程序中，而且也非常容易删除。如果输出非常复杂的信息字符串，觉得跟踪点非常讨厌，只需单击表示其位置的红色菱形，就可以删除跟踪点。

跟踪点的一个优点是允许方便地添加额外信息，如\$FUNCTION会把当前的函数名添加到输出信息中。这个信息可以用Debug和Trace命令来编写，但比较难。总之，输出调试信息的两种方法是：

- 诊断输出：总是要从应用程序中输出调试结果时使用这种方法，尤其是在要输出的字符串比较复杂，涉及几个变量或许多信息的情况下，使用该方法比较好。另外，如果要在执行发布版本的应用程序的过程中进行输出，Trace命令经常是唯一选择。
- 跟踪点：调试应用程序时，如果希望快速输出重要信息，以便消除语义错误，应使用跟踪点。

7.1.2 中断模式下的调试

本章描述的剩余调试技术在中断模式下工作。可以通过几种方式进入这种模式，这些方式都会以某种方式暂停程序的执行。

1. 进入中断模式

进入中断模式的最简单方式是在运行应用程序时，单击IDE中的

Pause按钮。这个Pause按钮在Debug工具栏上，你应把该工具栏添加到VS默认显示的工具栏中。为此，右击工具栏区域，然后选择Debug，这个工具栏如图7-6所示。



图7-6

在这个工具栏上，前3个按钮可以手工控制中断。在图7-6上，它们显示为灰色，因为在程序没有运行时，它们是不能工作的。在后面的章节需要其他按钮时，再介绍它们。

运行一个应用程序时，工具栏如图7-7所示。



图7-7

现在，就可以使用之前显示为灰色的3个按钮了。它们可以：

- 暂停应用程序的执行，进入中断模式
- 完全停止应用程序的执行（不进入中断模式，而是退出应用程序）
- 重新启动应用程序

暂停应用程序是进入中断模式的最简单方式，但这并不能更好地控制停止程序运行的位置。我们可能会停止在应用程序正常暂停的地方，例如，要求用户输入信息。还可以在长时间的操作或循环过程中进入中断模式，但停止的位置可能相当随机。一般情况下，最好使用断点。

断点

断点是源代码中自动进入中断模式的标记。它们可以配置为：

- 遇到断点时，立即进入中断模式
- 遇到断点时，如果布尔表达式的值为true，就进入中断模式
- 遇到某断点一定的次数后，进入中断模式
- 在遇到断点时，如果自从上次遇到断点以来变量的值发生了变化，就进入中断模式

注意，上述功能仅能用于调试程序。如果编译发布程序，将忽略所有断点。

添加断点有几种方法。要添加简单断点，当遇到该断点所在的代码行时，就中断执行，可以单击该代码行左边的灰色区域。其他方法包括：右击该代码行，选择Breakpoint|Insert Breakpoint菜单项；选择Debug|Toggle Breakpoint；或者按下F9键。

断点在代码行的旁边显示为一个红色圆圈，而该行代码也突出显示，如图7-8所示。

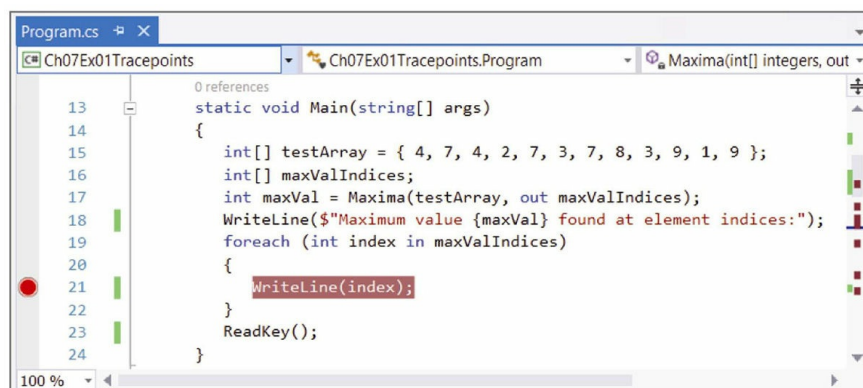


图7-8

使用Breakpoints窗口还可以查看文件中的断点信息（前面介绍过启用该窗口的方法）。在Breakpoints窗口中，可以禁用断点（删除描述信息左边的记号；禁用的断点用未填充的红色圆圈来表示）、删除断点、编辑断点的属性。还可以为断点添加标签，这是分组选定断点的一种便捷方式。可以在Labels列中查看标签，以及按标签过滤Breakpoints窗口中的项。

这个窗口中显示的Condition和Hit Count列是最有用的两个列。右击断点（在代码或Breakpoints窗口中），选择Condition或Hit Count菜单项，就可以编辑它们。

选择Condition将弹出一个对话框。在该对话框中可以键入任意布尔表达式，该表达式可以包含在断点位置仍在作用域内的任何变量。例如，可配置一个断点，输入表达式`maxVal>4`，选择Is true选项，在遇到这个断点且`maxVal`的值大于4时，就会触发该断点。还可以检查这个表达式是否有变化，仅当发生变化时，才会触发断点（例如，如果在遇到断点时，`maxVal`的值从2改为6，就会触发该断点）。

选择Hit Count将弹出另一个对话框。在这个对话框中可以指定在遇到断点多少次后才触发该断点。该对话框中的下拉列表提供了如下选项：

- 总是中断
- 在Hit Count等于多少次时中断
- 在Hit Count是某个数的倍数时中断
- 在Hit Count大于等于多少次时中断

所选的选项与在选项旁边的文本框中输入的值共同确定断点的行为。这个计数在比较长的循环中很有用，例如，在执行了前5000次循环后需要中断。如果不这么做，中断并重启5000次是很痛苦的。

进入中断模式的其他方式

进入中断模式还有两种方式。一种是在抛出一个未处理的异常时选择进入该模式。这种方式在本章后面讨论到错误处理时论述。另一种方式是在生成一条判定语句（**assertion**）时中断。

判定语句是可以用用户定义的消息中断应用程序的指令。它们常常用于应用程序的开发过程，作为测试程序能否平滑运行的一种方式。例如，在应用程序的某一处要求给定的变量值小于10，此时就可以使用一条判定语句，确定它是否为**true**，如果不是，就中断程序的执行。当遇到判定语句时，可以选择**Abort**，终止应用程序的执行；也可以选择**Retry**，进入中断模式；还可以选择**Ignore**，让应用程序像往常一样继续执行。

与前面的调试输出函数一样，判定函数也有两个版本：

- **Debug.Assert()**
- **Trace.Assert()**

其调试版本也是仅用于编译调试程序。

这两个函数带3个参数。第一个参数是一个布尔值，其值为**false**会触发判定语句。第二、第三个参数是两个字符串，分别把信息写到弹出的对话框和**Output**窗口中。上面的示例需要一个函数调用，如下所示：

```
Debug.Assert(myVar<10, "myVar is 10 or greater.",  
             "Assertion occurred in Main().");
```

判定语句通常在应用程序的早期使用比较有效。可以分发应用程序的一个发布程序，其中包含Trace.Assert()函数，以了解应用程序的运行情况。如果触发了判定语句，用户就会收到通知，把这些消息传递给开发人员。这样，即使开发人员不知道错误是如何发生的，也可以改正这个错误。

例如，在第一个字符串中提供有关错误的简短描述，在第二个字符串中提供下一步该如何操作的指示：

```
Trace.Assert(myVar<10, "Variable out of bounds.",  
            "Please contact vendor with the error code KCW001.");
```

如果触发了这条判定语句，用户将看到如图7-9所示的对话框。

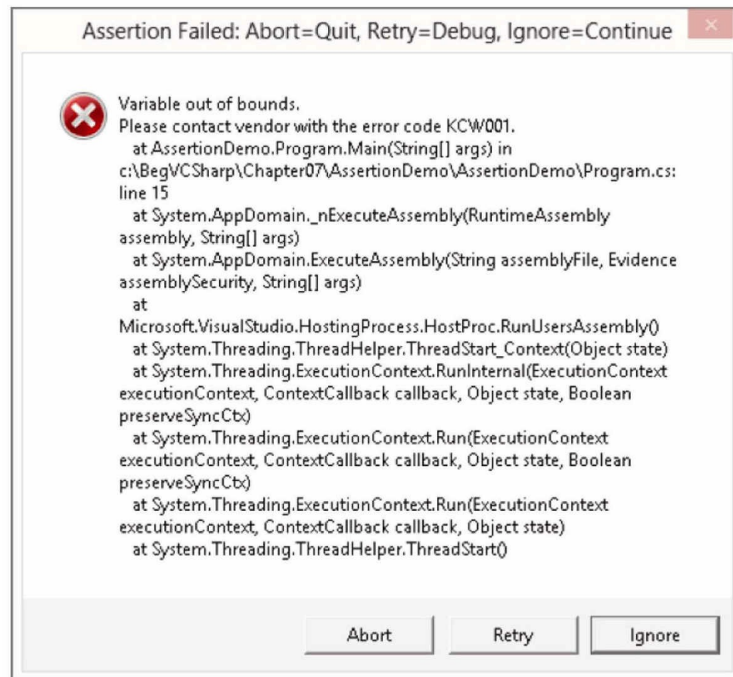


图7-9

诚然，这并不是最友好的对话框，因为它包含了许多令人感到迷惑的信息。但如果用户给开发人员发送了错误的屏幕图，开发人员就可以很快找出问题所在。

下一个要论述的主题是应用程序中断，以及进入中断模式后，我们可以做什么。一般情况下，进入中断模式的目的是找出代码中的错误（或确信程序工作正常）。一旦进入中断模式，就可以使用各种技巧分析代码，并分析应用程序在暂停时的状态。

2. 监视变量的内容

监视变量的内容是VS帮助我们使工作变得简单的一个例子。查看变量值的最简单方式是在中断模式下，使鼠标指向源代码中的变量名，此时会出现一个工具提示，显示该变量的信息，其中包括该变量的当前值。

还可以高亮显示整个表达式，以相同方式得到该表达式的结果。对于比较复杂的值（例如数组），甚至可以扩展工具提示中的值，查看各个数组元素项。

甚至可以把这些工具提示窗口固定到代码视图中，这对于查看特别感兴趣的变量很有帮助。固定的工具提示会一直显示，所以即使在停止并重启调试后，仍然可以看到它们。甚至可以在固定的工具提示中添加注释，移动工具提示窗口，查看变量的最后一个值，即使应用程序并没有运行也同样如此。

注意，在运行应用程序时，IDE中各个窗口的布局发生了变化。默认情况下，运行期间会发生如下变化（变化的情况因具体的安装而异）：

- Properties窗口和其他一些窗口会消失，其中可能包括Solution Explorer窗口
- Error List窗口会被IDE窗口底部的两个新窗口替代
- 新窗口中会出现几个新的选项卡

新的屏幕布局如图7-10所示。这可能与读者的显示情况不完全相同，一些选项卡和窗口可能不完全匹配。但是，这些窗口的功能（后面将讨论）是相同的，这个显示完全可以通过View和Debug|Windows菜单来定制（在中断模式下），也可以在屏幕上拖动窗口，重新设定它们的位置。

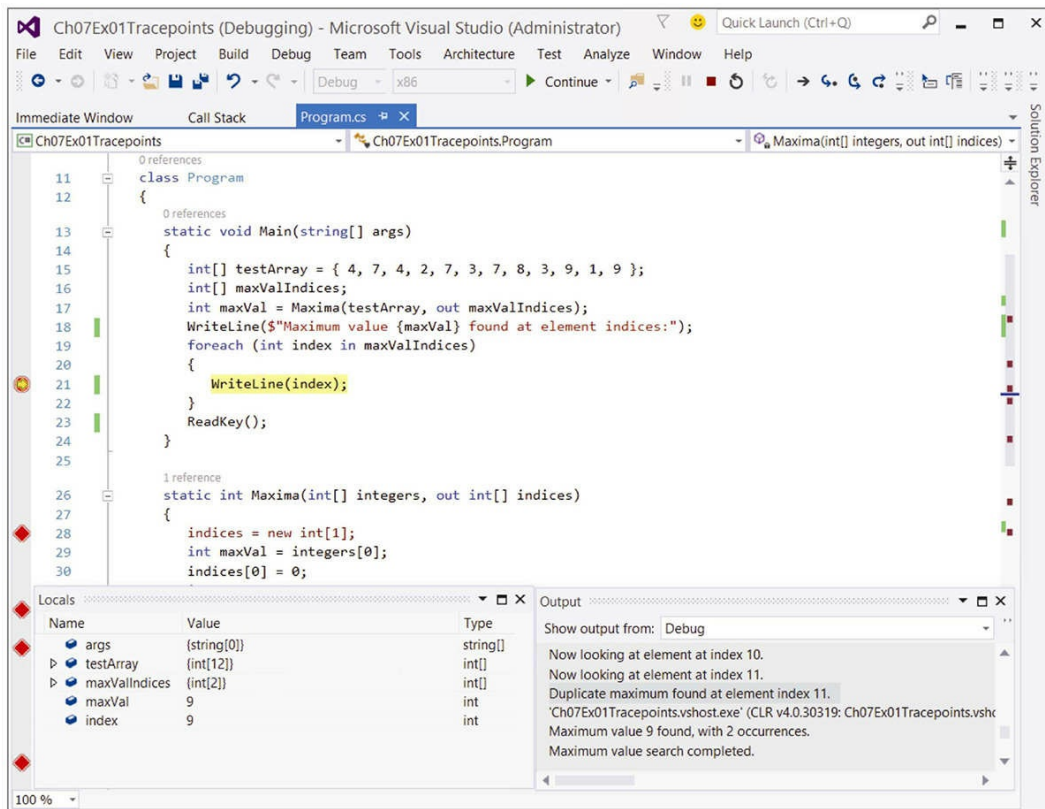
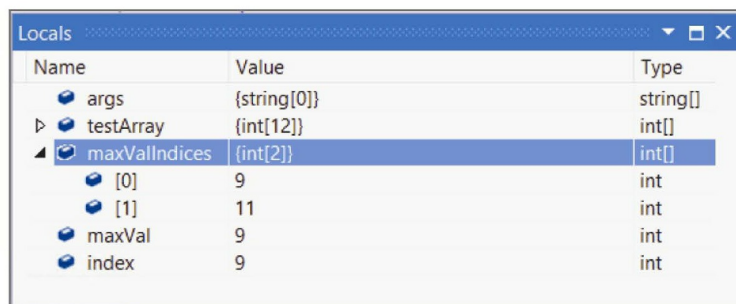


图7-10

左下角的新窗口在调试时非常有用，它允许在中断模式下，密切监视应用程序的变量值。它包含3个选项卡，如下所示：

- Autos——当前和前面的语句使用的变量（Ctrl+D, A）
- Locals——作用域内的所有变量（Ctrl+D, L）
- Watch N——可定制的变量和表达式显示（其中N为1-4的值，在Debug|Windows|Watch上）

这些选项卡的工作方式或多或少有些类似，并根据它们的特定功能添加了各种附加特性。一般情况下，每个选项卡都包含一个变量列表，其中包括变量的名称、值和类型等信息。更复杂的变量（如数组）可以使用变量名左边的+和-（展开/折叠）符号进一步查看，它们的内容可以树状视图的方式显示。例如，在前面的示例中，在代码中放置了一个断点，得到的Locals选项卡如图7-11所示，其中显示了数组变量maxValIndices的展开视图。



Name	Value	Type
args	{string[0]}	string[]
testArray	{int[12]}	int[]
maxValIndices	{int[2]}	int[]
[0]	9	int
[1]	11	int
maxVal	9	int
index	9	int

图7-11

在这个视图中，还可以编辑变量的内容。它有效地绕过了前面代码中的其他变量赋值。为此，只需要在Value列中为要编辑的变量输入一

一个新值即可。也可以将这种技巧用于其他情况，例如，需要修改代码才能编辑变量值的情况。

可通过Watch窗口监视特定变量或涉及特定变量的表达式。要使用这个窗口，只需在Name列中键入变量名或表达式，就可以查看它们的结果。注意，并不是应用程序中的所有变量在任何时候都在作用域内，并在Watch窗口中对变量做出标记。例如，图7-12显示了一个Watch窗口，其中包含几个示例变量和表达式，在遇到Maxima()函数末尾前面的一个断点时，会显示这个Watch窗口。

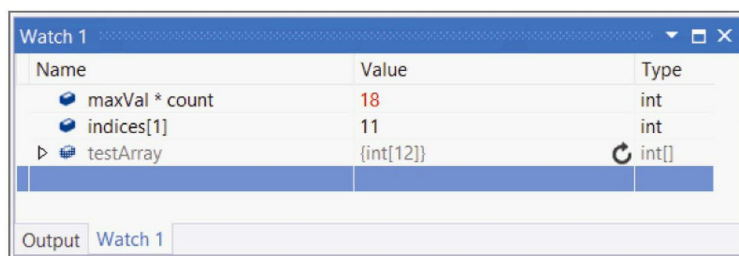


图7-12

testArray数组对于Main()来说是局部数组，所以在该图中没有值，它是灰显的。

3. 单步执行代码

前面介绍了如何在中断模式下查看应用程序的运行情况，下面讨论如何在中断模式下使用IDE单步执行代码，查看代码的准确执行结果。人们的思维速度不会比计算机运行得更快，所以这是一个极有价值的技巧。

VS进入中断模式后，在代码视图的左边，马上要执行的代码旁边会出现一个黄色箭头光标（如果使用断点进入中断模式，该光标最初应显示在断点的红色圆圈中），如图7-13所示。

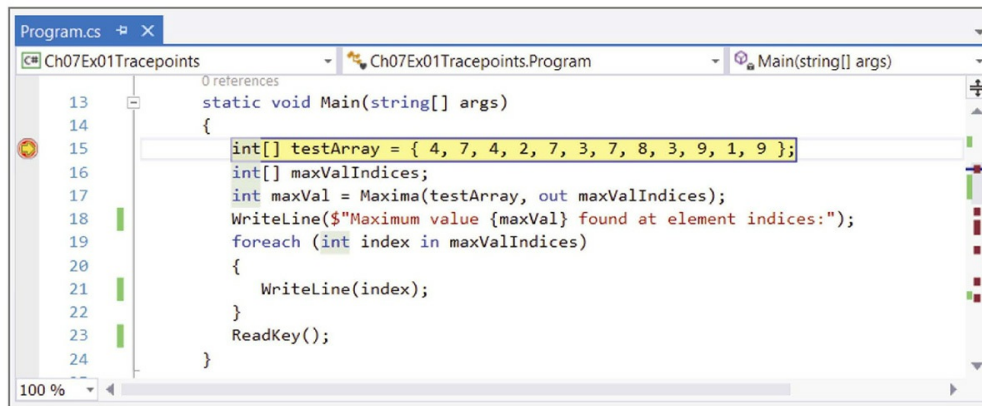


图7-13

这显示了在进入中断模式时程序执行到的位置。在这个位置，可以选择逐行执行。为此，使用前面看到的其他一些Debug工具栏按钮，如图7-14所示。



图7-14

第6、第7、第8个图标控制了中断模式下的程序流。它们依次是：

- Step Into——执行并移动到下一条要执行的语句上
- Step Over——同上，但不进入嵌套的代码块，包括函数
- Step Out——执行到代码块的末尾处，在执行完该语句块后，重新进入中断模式

如果要查看应用程序执行的每个操作，可以使用**Step Into**按顺序执行指令，这包括在函数中执行，如上面示例中的**Maxima()**。当光标到达第17行，调用**Maxima()**时，单击这个图标，会使光标移动到**Maxima()**函数内部的第一行代码上。而如果光标移到第17行时单击**Step Over**，就会使光标移动到第18行，不进入**Maxima()**中的代码（但仍执行这段代码）。如果单步执行到不感兴趣的函数，可以单击**Step Out**，返回到调用该函数的代码。在单步执行代码时，变量的值可能会发生变化。注意观察上一节讨论的**Watch**窗口，可以看到变量值的变化情况。

通过右击代码行并选择**Set Next Statement**，或将黄色箭头拖到不同的代码行，也可以更改接下来要执行的代码行。这有时是不可行的，例如跳过变量初始化时。但是，当跳过存在问题的代码行来查看发生的情况时，或向后移动箭头来重复执行代码时，这种方法是非常有用的。

在存在语义错误的代码中，这些技巧也许是最有效的。可以单步执行代码，当执行到有错误的代码时，错误会像正常运行程序那样发生。或者可以修改执行代码，让语句多次执行。在这个过程中，可以监视数据，看看什么地方出错。本章后面将使用这个技巧查看示例应用程序的执行情况。

4. **Immediate**和**Command**窗口

通过**Command**和**Immediate**窗口（在**Debug**窗口菜单下），可以在运行应用程序的过程中执行命令。通过**Command**窗口可以手动执行VS操作（例如，菜单和工具栏操作），**Immediate**窗口可以执行与当前正在执行的源代码不同的额外代码，以及计算表达式。

VS中的这些窗口在内部是链接在一起的。甚至可以在它们之间切换：输入命令immed，可以从Command窗口切换到Immediate窗口；输入cmd可以从Immediate窗口切换到Command窗口。

下面详细讨论Immediate窗口，因为Command窗口仅适用于复杂的操作。Immediate窗口最简单的用法是计算表达式，有点像Watch窗口中的一次性使用。为此，只需要键入一个表达式，并按回车键即可。接着就会显示请求的信息，如图7-15所示。

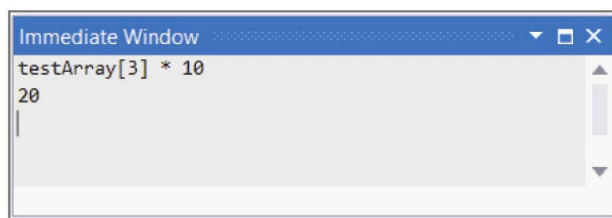


图7-15

可以在这里修改变量的内容，如图7-16所示。

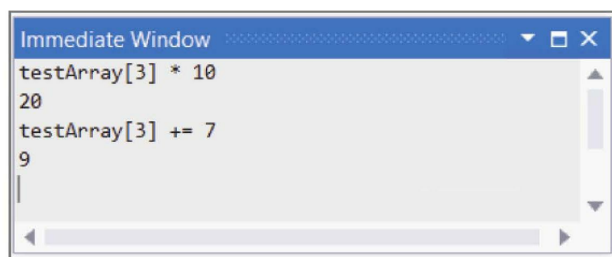


图7-16

大多数情况下，使用前面介绍的变量监视窗口更容易得到相同的效果，但这个技巧对于调整变量值和测试表达式很方便。

5. Call Stack窗口

这是最后一个要讨论的窗口，它描述了程序是如何执行到当前位置的。简言之，该窗口显示了当前函数、调用它的函数以及调用该函数的函数（即一个嵌套的函数调用列表）。调用的确切位置也被记录下来。

在前面的示例中，在执行到Maxima()时进入中断模式，或者使用代码单步执行功能移动到这个函数的内部，得到如图7-17所示的信息。

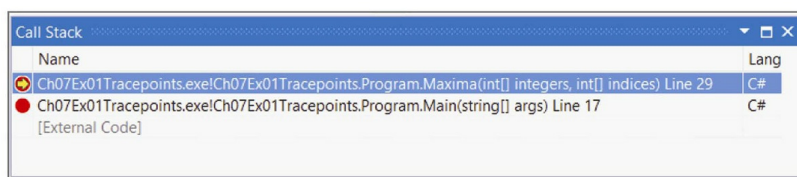


图7-17

如果双击某一项，就会移动到相应的位置，跟踪代码执行到当前位置的过程。第一次检测错误时，这个窗口非常有用，因为它们可以查看临近错误发生时的情况。对于常用函数中出现的错误，这有助于找到错误的源头。

7.2 错误处理

本章的第一部分讨论如何在应用程序的开发过程中查找和改正错误，使这些错误不会在发布的代码中出现。但有时，我们知道可能会有错误发生，但不能100%地肯定它们不会发生。此时，最好能预料到错误的发生，编写足够健壮的代码以处理这些错误，而不必中断程序的执行。

错误处理就是用于这个目的。本节将介绍异常和处理它们的方式。异常是在运行期间代码中产生的错误，或者由代码调用的函数产生的错误。这里的“错误”定义要比以前更含糊，因为异常可能是在函数等结构中手工产生的。例如，如果函数的一个字符串参数不是以a开头，就产生一个异常。严格来讲，从该函数的外部看这并不是一个错误，但调用该函数的代码会把它看成错误。

在本书前面已经遇到几次异常了。最简单的示例是试图定位一个超出范围的数组元素，例如：

```
int[] myArray = { 1, 2, 3, 4 };  
int myElem = myArray[4];
```

这会产生如下异常信息，并中断应用程序的执行：

```
Index was outside the bounds of the array.
```

异常在名称空间中定义，大多数异常的名称清晰地说明了它们的用途。在这个示例中，产生的异常称为

`System.IndexOutOfRangeException`，说明我们提供的`myArray`数组索引不在允许使用的索引范围内。只有在异常未处理时，这个信息才会显示出来，应用程序也才会中断执行。下一节将讨论如何处理异常。

7.2.1 try...catch...finally

C#语言包含结构化异常处理（Structured Exception Handling, SEH）的语法。用3个关键字可以标记出能处理异常的代码和指令，如果发生异常，就使用这些指令处理异常。用于这个目的的3个关键字是`try`、`catch`和`finally`。它们都有一个关联的代码块，必须在连续的代码行中使用。其基本结构如下：

```
try
{
    ...
}
catch (<exceptionType>
    e) when (filterIsTrue)
{
    <await methodName(e);>
    ...
}
finally
{
```

```
<await method name>

...

}
```

也可以在`catch`或`finally`块内使用C# 6引入的`await`。`await`关键字用于支持先进的异步编程技术，避免瓶颈，且可以提高应用程序的总体性能和响应能力。利用`async`和`await`关键字的异步编程在本书中不讨论；然而，这些关键字简化了这个编程技术的实现，所以强烈建议学习它们。

也可以只有`try`块和`finally`块，而没有`catch`块，或者有一个`try`块和好几个`catch`块。如果有一个或多个`catch`块，`finally`块就是可选的，否则就是必需的。这些代码块的用法如下：

- `try`——包含抛出异常的代码（在谈到异常时，C#语言用“抛出”这个术语表示“生成”或“导致”）。
- `catch`——包含抛出异常时要执行的代码。`catch`块可以使用`<exceptionType>`，设置为只响应特定的异常类型（如`System.IndexOutOfRangeException`），以便提供多个`catch`块。还可以完全省略这个参数，让通用的`catch`块响应所有异常。C# 6引入了一个概念“异常过滤”，通过在异常类型表达式后添加`when`关键字来实现。如果发生了该异常类型，且过滤表达式是`true`，就执行`catch`块中的代码。
- `finally`——包含始终会执行的代码，如果没有产生异常，则在`try`块之后执行，如果处理了异常，就在`catch`块后执行，或者在未处理的异常“上移到调用堆栈”之前执行。“上移到调用堆栈”表示，SEH允许嵌套`try...catch...finally`块，可以直接嵌套，也可以在`try`块包含的函数调用中嵌套。例如，如果在被调用的函数中没有`catch`块能处理某个异常，就由调用代码中的`catch`块处理。如果始终没有匹配的

catch块，就终止应用程序。finally块在此之前处理正是其存在的意义，否则也可以在try...catch...finally结构的外部放置代码。

在try块的代码中出现异常后，依次发生的事件如下，如图7-18所示：

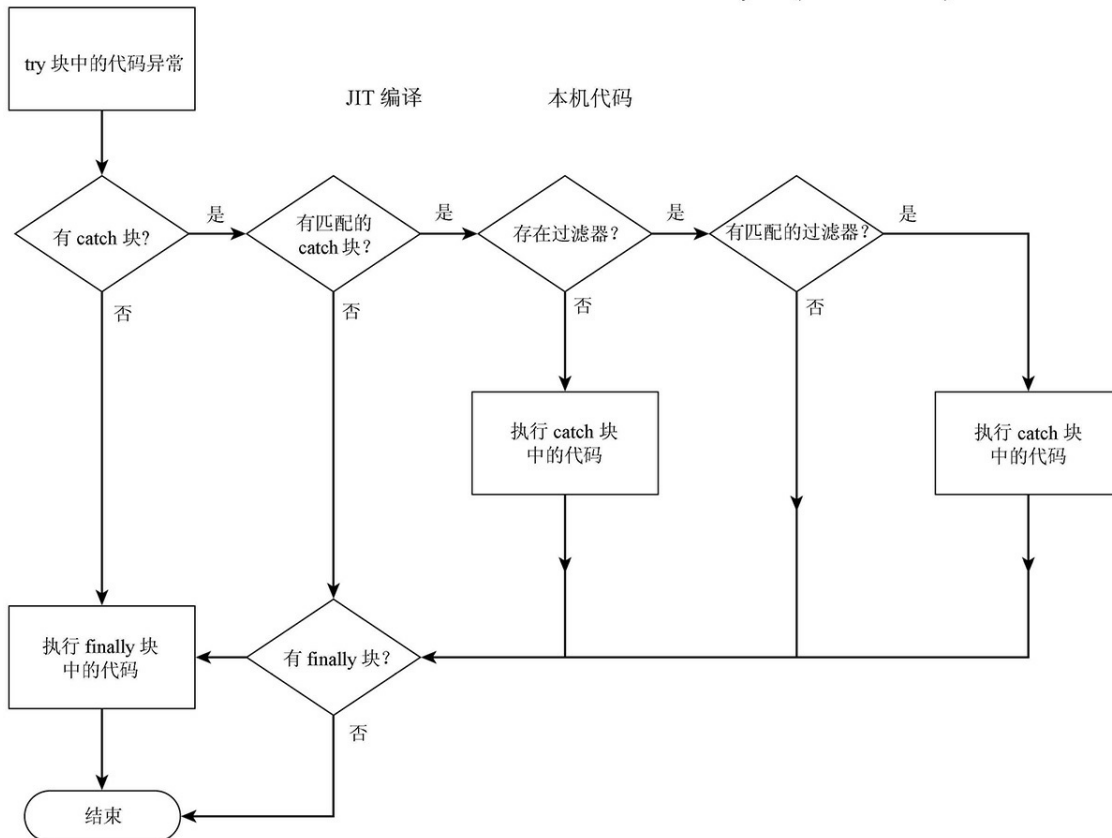


图7-18

- try块在发生异常的地方中断程序的执行。
- 如果有catch块，就检查该块是否匹配已抛出的异常类型。如果没有catch块，就执行finally块（如果没有catch块，就一定要有finally块）。
- 如果有catch块，但它与已发生的异常类型不匹配，就检查是否有其

他catch块。

- 如果有catch块匹配已发生的异常类型，且有一个异常过滤器是true，就执行它包含的代码，再执行finally块（如果有的话）。
- 如果有catch块匹配已发生的异常类型，但没有异常过滤器，就执行它包含的代码，再执行finally块（如果有的话）。
- 如果catch块都不匹配已发生的异常类型，就执行finally块（如果有的话）。

注意： 如果存在两个处理相同异常类型的catch块，就只执行异常过滤器为true的catch块中的代码。如果还存在一个处理相同异常类型的catch块，但没有异常过滤器或异常过滤器是false，就忽略它。只执行一个catch块的代码，catch块的顺序不影响执行流。

下面用一个示例来说明异常处理。这个示例以几种方式抛出和处理异常，以便读者了解其机制。

试一试：异常处理：**Ch07Ex02\Program.cs**

（1）在C:\BegVCSharp\Chapter07目录中创建一个新的控制台应用程序Ch07Ex02。

（2）修改代码，如下所示（这里显示的行号注释有助于将代码与后面讨论的内容联系起来，在本章的可下载代码中也包含这些行号，以

方便参考)：

```
class Program
{
    static string[] eTypes = { "none", "simple", "index",

                                "nested index", "filter" };

    static void Main(string[] args)
    {
        foreach (string eType in eTypes)

        {

            try

            {
```

```
WriteLine("Main() try block reached."); // Line 21
```

```
WriteLine($"ThrowException(\"{eType}\") called.");
```

```
ThrowException(eType);
```

```
WriteLine("Main() try block continues."); // Line 23
```

```
}
```

```
catch (System.IndexOutOfRangeException e) when (eType =
```

```
{
```

```
WriteLine("Main() FILTERED System.IndexOutOfRangeException
```

```
$"catch block reached. Message:\n\"{e.Message}
```

```
}
```

```
catch (System.IndexOutOfRangeException e)           // Line
```

```
{
```

```
WriteLine("Main() System.IndexOutOfRangeException cat
```

```
$"block reached. Message:\n\"{e.Message}\"");
```

```
}
```

```
catch // Line 36
```

```
{
```

```
WriteLine("Main() general catch block reached.");
```

```
}
```

```
finally
```

```
{
```

```
    WriteLine("Main() finally block reached.");
```

```
}
```

```
WriteLine();
```

```
}
```

```
ReadKey();
```

```
}
```

```
static void ThrowException(string exceptionType)
```

```
{
```

```
    WriteLine($"ThrowException(\"{exceptionType}\") reached.'
```

```
    switch (exceptionType)
```

```
{
```

```
    case "none":
```

```
        WriteLine("Not throwing an exception.");
```

```
break; // Line 57
```

```
case "simple":
```

```
WriteLine("Throwing System.Exception.");
```

```
throw new System.Exception(); // Line 60
```

```
case "index":
```

```
WriteLine("Throwing System.IndexOutOfRangeException."
```

```
eTypes[5] = "error"; // Line 63
```



```
break;
```

```
case "nested index":
```

```
try // Line 66
```

```
{
```

```
WriteLine("ThrowException(\"nested index\") " +
```

```
"try block reached.");
```

```
WriteLine("ThrowException(\"index\") called.");
```

```
ThrowException("index");           // Line 71
```

```
}
```

```
catch                               // Line 73
```

```
{
```

```
WriteLine("ThrowException(\"nested index\") generated
```

```
+ " catch block reached.");
```

```
}
```

```
finally
```

```
{
```

```
WriteLine("ThrowException(\"nested index\") finally
```

```
+ " block reached.");
```

```
}
```

```
break;
```

```
case "filter":
```

```
try // Line 86
```

```
{
```

```
WriteLine("ThrowException(\"filter\") " +
```

```
"try block reached.");
```

```
WriteLine("ThrowException(\"index\") called.");
```

```
ThrowException("index"); // Line 91
```

```
}
```

```
catch // Line 93
```

```
{
```

```
WriteLine("ThrowException(\"filter\") general"
```

```
+ " catch block reached.");
```

```
}
```

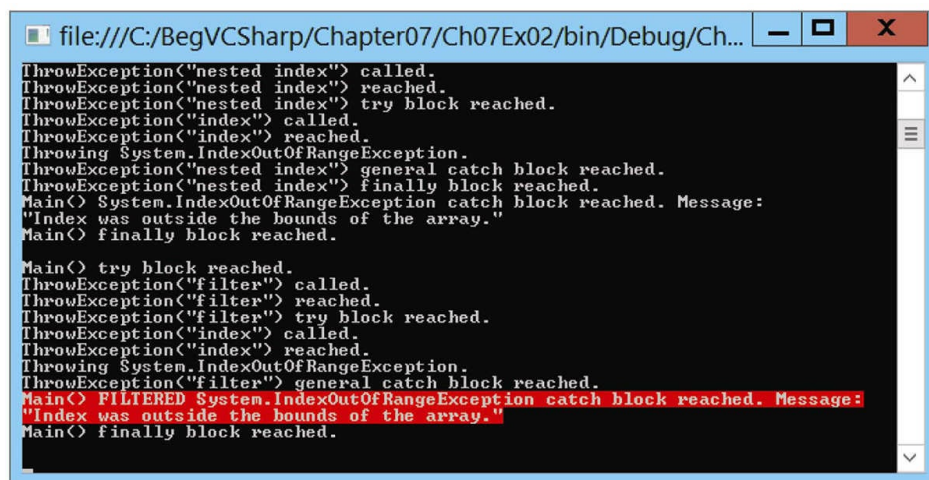
```
break;
```

```
}
```

```
}
```

```
}
```

(3) 运行应用程序，结果如图7-19所示。



```
file:///C:/BegVCSharp/Chapter07/Ch07Ex02/bin/Debug/Ch...
ThrowException("nested index") called.
ThrowException("nested index") reached.
ThrowException("nested index") try block reached.
ThrowException("index") called.
ThrowException("index") reached.
Throwing System.IndexOutOfRangeException.
ThrowException("nested index") general catch block reached.
ThrowException("nested index") finally block reached.
Main() System.IndexOutOfRangeException catch block reached. Message:
"Index was outside the bounds of the array."
Main() finally block reached.

Main() try block reached.
ThrowException("filter") called.
ThrowException("filter") reached.
ThrowException("filter") try block reached.
ThrowException("index") called.
ThrowException("index") reached.
Throwing System.IndexOutOfRangeException.
ThrowException("filter") general catch block reached.
Main() FILTERED System.IndexOutOfRangeException catch block reached. Message:
"Index was outside the bounds of the array."
Main() finally block reached.
```

图7-19

示例说明

这个应用程序在`Main()`中有一个`try`块，它调用函数`ThrowException()`。这个函

- `ThrowException ("none")` —不抛出异常。
- `ThrowException ("simple")` —生成一般异常。
- `ThrowException ("index")` —生成`System.IndexOutOfRangeException`。
- `ThrowException ("nested index")` —包含它自己的`try`块，其中的代码调用`ThrowException ("nested index")`。
- `ThrowException ("filter")` —包含自己的`try`块，`try`块包含的代码调用`ThrowException ("filter")`。

其中的每个`string`参数都存储在全局数组`eTypes`中，在`Main()`函数中迭代，用每

注意：

上面代码清单的步骤（2）未列出两条`throw`语句。这两条语句在`ThrowException`（

在代码的第21行添加一个新断点（用默认的属性），该行代码如下：

```
WriteLine("Main() try block reached.");
```

注意：

这里使用了本章可下载代码中的行号来表示代码。如果关闭了行号，可以选择`Tools | C`

在调试模式下运行应用程序。程序立即进入中断模式，此时光标停在第20行上。如果

执行到`ThrowException()`函数后，Locals窗口会发生变化。`eType`和`args`超出了

接着处理`finally`块。再单击`Step Into`几次，执行完`finally`块和`foreach`的第

继续使用`Step Into`单步执行`ThrowException()`，最终会执行到第60行：

```
throw new System.Exception();
```

这里使用C#的`throw`关键字生成一个异常，需要为这个关键字提供新初始化的异常作

注意：

在`case`块中使用`throw`时，不需要`break`语句，使用`throw`就可以结束该块的执行。

在使用`Step Into`执行这条语句时，将从第36行开始执行一般的`catch`块。因为与第

这次`ThrowException()`在第63行生成一个异常：

```
eTypes[5] = "error";
```

`eTypes`是一个全局数组，所以可以在这里访问它。但是这里试图访问数组中的第6个

这次`Main()`中有多个匹配的`catch`块，其中第26行的一个`catch`块有异常过滤器（

单步执行到下一个`catch`块，从第32行开始。这个块中调用的`WriteLine()`使用`e.`

在执行到`ThrowException()`中的`switch`结构时，进入一个新的`try`块，从第66行

继续单步执行代码，这次到达`ThrowException()`中的`switch`结构时，进入一个新

与前面的异常处理一样，现在单步执行这个`catch`块，以及关联的`finally`块，最后

7.2.2 列出和配置异常

.NET Framework包含许多异常类型，可以在代码中自由抛出和处理这些类型的异常

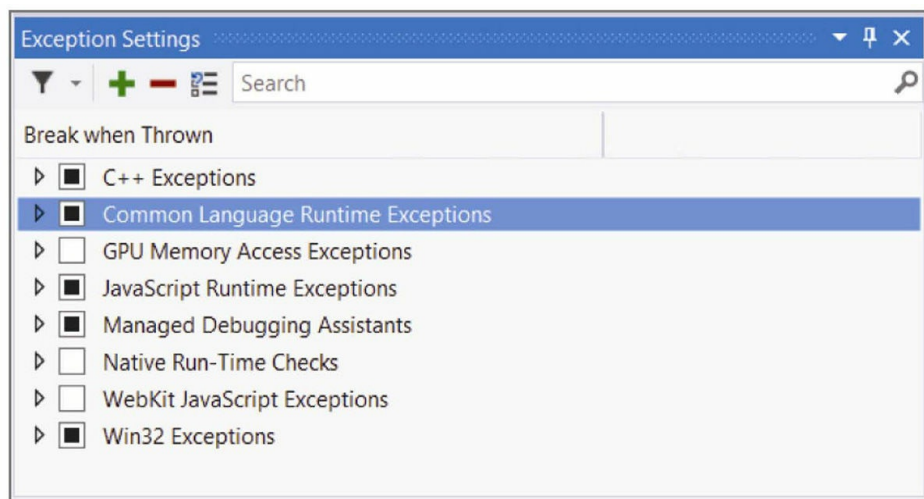


图7-20

该对话框按照类别和.NET库名称空间列出异常。展开Common Language Runtime

每个异常都可以使用右边的复选框来配置。使用（break when）Thrown时，即使是

7.3 练习

(1) “使用`Trace.WriteLine()`要优于使用`Debug.WriteLine()`，因为调试版本`Trace.WriteLine()`在发布版本中也会被编译进去。”对吗？

(2) 为一个简单的应用程序编写代码，其中包含一个循环，该循环在运行5000次后

(3) “只有在不执行`catch`块的情况下，才执行`finally`代码块”，对吗？

(4) 下面定义了一个枚举数据类型`orientation`。编写一个应用程序，使用结构化

```
enum Orientation : byte
{
    North = 1,
    South = 2,
    East = 3,
    West = 4
}
myDirection = checked((Orientation)myByte);
```

附录A给出了练习答案。

7.4 本章要点

主题	要点
错误类型	编译期间的语法错误和运行期间的致命错误都会使应用程序完全失败，语义错误（或逻辑错误）比较微妙，可能会使应用程序执行不正确，或者以未预料到的方式执行
输出调试信息	我们可以编写代码，把有帮助的信息输出到Output窗口中，以帮助在IDE中进行调试。为此需要使用Debug和Trace系列函数，其中Debug函数在发布版本中会被忽略。对于投入生产的应用程序，应将调试输出写入日志文件。另外，还可以使用跟踪点输出调试信息
中断模式	可以通过断点、判定语句，或者在发生未处理的异常时，手工进入中断模式（实际上就是暂停应用程序的状态）。可以在代码的任意位置添加断点，还可以把断点配置为仅在特定条件下中断执行。在中断模式下，可以检查变量的内容（使用各种调试信息窗口），每次执行一行代码，以帮助确定哪里出现了错误
异常	异常是运行期间发生的错误，可以通过编程方式捕获和处理这种错误，以防应用程序终止。调用函数或处理变量时，可能会发生许多不同类型的异常。还可以使用throw关键字生成异常
异常处理	代码中未处理的异常会使应用程序终止。使用try、catch和finally代码块处理异常。try块标记了一个启用异常处理的代码段，catch块包含的代码仅在异常发生时执行，它可以匹配特定类型的异常，还可以包含多个catch块。finally块指定异常处理完毕后执行的代码，如果没有发生异常，finally块就指定在try块执行完毕后执行的代码。只能包含一个finally块，如果包含了catch块，finally块就是可选的

第8章 面向对象编程简介

本章内容：

- 什么是面向对象编程
- OOP技术
- 桌面应用程序对OOP的依赖关系

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 8 Code后，可以找到与本章示例对应的单独文件。

本书前面介绍了C#语法和编程的所有基础知识，以及调试应用程序的方法。现在我们已经可以编写出可供使用的控制台应用程序了。但是，要了解C#语言和.NET Framework的强大功能，还需要使用面向对象编程（Object-Oriented Programming, OOP）技术。实际上，前面已经使用了这些技术，但为了使学习任务简单一些，在列出代码示例时没有重点讲述该技术。

本章先不考虑代码，而主要探讨OOP的基本原理。OOP会很快把我们领回C#语言，因为它与OOP是一种共生关系。本章介绍的所有概念在

后续章节中都会再次讨论，并用演示性的代码来说明。所以，如果你在第一次阅读本章时没有掌握所有内容，不必惊慌。

本章首先介绍OOP的基础知识，包括回答最基本的问题“什么是对象？”。很快你就会发现许多OOP术语在一开始很难理解，但本章提供了大量的解释。使用OOP需要以另一种方式来看待编程。

除了讨论OOP的一般原理外，本章还将进入一个需要深刻理解OOP的领域：桌面应用程序。此类应用程序依赖Windows环境，使用诸如菜单、按钮等特性，有许多值得描述的地方，在Windows环境中可以有效地说明OOP要点。

8.1 面向对象编程的含义

面向对象编程解决了传统编程技巧的许多问题。前面介绍的编程方法称为函数（或过程）化编程，常会导致所谓的单一应用程序，即所有功能都包含在几个代码模块（常常是一个代码模块）中。而使用OOP技术，常常要使用许多代码模块，每个模块都提供特定功能。而且，每个模块都是孤立的，甚至与其他模块完全独立。这种模块化编程方法提供了非常大的多样性，大大增加了重用代码的机会。

为进一步说明这个问题，把计算机上的一个高性能应用程序想象成一辆一流赛车。如果使用传统的编程技巧，这辆赛车就是一个单元。如果要改进这辆车，就必须替换整车，把它送回厂商那里，让汽车专家升级它，或者购买一辆新车。如果使用OOP技术，就只需要从厂商处购买新的引擎，自己按照其说明替换它，而不必用钢锯切割车体。

在传统应用程序中，执行流常是简单的、线性的。把应用程序加载到内存中，从A点开始执行，在B点结束，然后从内存中卸载，在这个过程中可能用到其他各种实体，例如存储介质上的文件或显卡的功能，但处理的主体总是位于一个地方。用到的代码一般与使用各种数学和逻辑方式处理数据相关。处理方法通常比较简单，使用基本的数据类型（例如整型和布尔值）建立比较复杂的数据表达方式。

而使用OOP，事情就不是这么直接了。尽管可以获得相同的效果，但其实现方式是完全不同的。OOP技术以结构、数据的含义以及数据和数据之间的交互操作为基础。这通常意味着要把更多精力放在项目的设计阶段，其好处是项目的可扩展性比较高。一旦对某种类型的数据的表

达方式达成一致，这种表达方式就会应用到应用程序以后的版本中，甚至是全新应用程序中。这种一致的表达方式可以极大地缩短开发时间。这就是上述赛车示例的工作原理。这里的一致是指“引擎”的代码是结构化的，这样就可以很容易地替换成新代码（即新引擎），而不需要找厂商帮忙。这也表示，引擎创建出来后可用于其他目的，可以把它安装到另一辆车上，或者用它驱动潜艇。

除了数据表达方式的一致性外，OOP编程还常可以简化任务，因为较抽象实体的结构和用法也是一致的。例如，不仅把输出结果发送给设备（如打印机）所使用的数据格式是一致的，而且与该设备交换数据的方法也是一致的，这包括它理解的指令等。回到赛车示例上，要达成的一致做法包括引擎如何连接到油箱，如何把驱动力传送给车轮等。

顾名思义，OOP技术要使用对象。

8.1.1 对象的含义

对象就是OOP应用程序的一个组成部件。这个组成部件封装了部分应用程序，这部分程序可以是一个过程、一些数据或一些更抽象的实体。

简单地说，对象非常类似于本书前面讨论的结构类型，包含变量成员和函数类型。它所包含的变量组成了存储在对象中的数据，其中包含的函数可以访问对象的功能。略为复杂的对象可能不包含任何数据，而只包含函数，表示一个过程。例如，可以使用表示打印机的对象，其中的函数可以控制打印机（允许打印文档、测试页等）。

C#中的对象是从类型中创建的，就像前面的变量一样。对象的类型在OOP中有一个特殊名称：类。可以使用类的定义实例化对象，这表示创建该类的一个命名实例。“类的实例”和对象的含义相同，但“类”和“对象”是完全不同的概念。

注意：术语“类”和“对象”常常混淆，从一开始就正确区分它们是非常重要的，使用前面的赛车示例有助于区分这两个术语。在这个示例中，类是指汽车的模板，或者用于构建汽车的规划。汽车本身是这些规划的实例，所以可以看成对象。

本章将使用统一建模语言（Unified Modeling Language, UML）语法研究类和对象。UML是为应用程序建模而设计的，从组成应用程序的对象，到它们执行的操作，到我们希望有的用例，应有尽有。这里只使用这门语言的基本部分，在使用它们的过程中进行解释，但不考虑比较复杂的部分，因为UML是一个很专业的主题，有很多图书专门介绍它。

图8-1是打印机类Printer的UML表示方法。类名显示在这个框的顶部（后面将论述下面两个区域）。

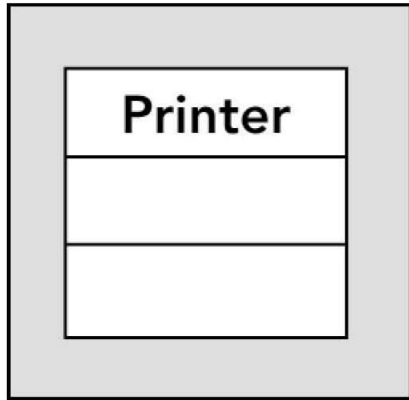


图8-1

图8-2是这个**Printer**类的一个实例**myPrinter**的UML表示方法。

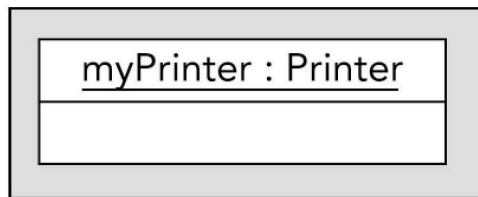


图8-2

在顶部，首先显示实例名，其后是类名。这两个名称用一个冒号分隔。

1. 属性和字段

可以通过属性和字段访问对象中包含的数据。这个对象数据可以用于区分不同的对象，因为同一个类的不同对象在属性和字段中存储了不同的值。

包含在对象中的不同数据构成了对象的状态。假定一个对象类表示

一杯咖啡，称为CupOfCoffee。在实例化这个类（即创建这个类的对象）时，必须提供对类有意义的状态。此时可以使用属性和字段，让代码能通过该对象设置要使用的咖啡品牌，咖啡中是否加牛奶或方糖，咖啡是否即溶等。于是，给定的这杯咖啡对象就有了指定的状态，例如，加牛奶和两块方糖的哥伦比亚过滤咖啡。

字段和属性都可以键入，所以可以把信息存储在字段和属性中，作为string值、int值等。但属性与字段是不同的，因为属性不提供对数据的直接访问。对象能让用户不考虑数据的细节，不需要在属性中用一对一的方式表示。如果在CupOfCoffee实例中使用一个字段表示方糖的数量，用户就可以在该字段中放置自己喜欢的值，其取值范围仅由存储该信息的类型来限制。例如，如果使用int来存储这个数据，用户就可以使用-2 147 483 648至2 147 483 647之间的任意值，如第3章所述。显然，并不是所有的值都有意义，尤其是负值，一些较大的正值将需要非常大的咖啡杯。但如果使用一个属性来表示，就可以限制这个值，例如介于0和2之间的一个数字。

一般情况下，在访问状态时最好提供属性而不是字段，因为这样可以更好地控制各种行为，这个选择不会影响使用对象实例的代码，因为使用属性和字段的语法是相同的。

对属性的读写访问也可以由对象来明确定义。某些属性是只读的，只能查看它们的值，而不能改变它们（至少不能直接改变）。这常常是同时读取几个状态的一个有效技巧。CupOfCoffee类有一个只读属性Description，在请求它时，就返回一个字符串，表示该类的一个实例的状态（例如前面给出的字符串）。也可以通过查看几个属性，把相同的数据组合起来，但这样的属性可以节省时间和精力。还可以有只写的属性，其操作方式是类似的。

除了对属性的读/写访问外，还可以为字段和属性指定另一种访问权限，称为可访问性。可访问性确定了什么代码可以访问这些成员，它们可用于所有代码（公共）还是只能用于类中的代码（私有），或者使用更复杂的模式（详见本章后面的内容）。常见的情况是把字段设置为私有，通过公共属性访问它们。这样，类中的代码就可以直接访问存储在字段中的数据，而公共属性禁止外部用户访问这些数据，以防他们在其中放置无效的内容。公共成员是类公开的成员。

要更清晰地阐明这个问题，可以把可访问性与变量的作用域等同起来。例如，私有字段和属性可以看成拥有它们的对象的局部成员，而公共字段和属性的作用域也包括对象以外的代码。

在类的UML表示方法中，用第二部分显示属性和字段，如图8-3所示。

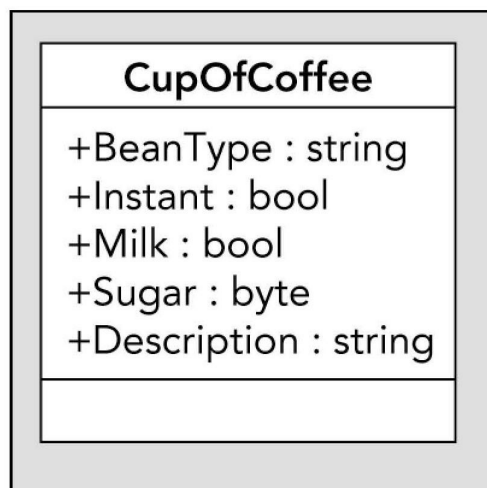


图8-3

这是CupOfCoffee类的表示方式，前面为它定义了5个成员（属性或字段，在UML中，它们没有区别）。每个成员都包含下述信息：

- 可访问性：+号表示公共成员，-号表示私有成员。但一般情况下，本章的图中不显示私有成员，因为这些信息是类内部的信息。至于读/写访问，则不提供任何信息。
- 成员名。
- 成员的类型。

冒号用于分隔成员名和类型。

2. 方法

“方法”这个术语用于表示对象中的函数。这些函数调用的方式与其他函数相同，使用返回值和参数的方式也相同（详见第6章）。

方法用于访问对象的功能。与字段和属性一样，方法也可以是公共的或私有的，按照需要限制外部代码的访问。它们通常使用对象的状态来影响它们的操作，在需要时访问私有成员，如私有字段。例如，CupOfCoffee类定义了一个方法AddSugar()，该方法对递增方糖数提供了比设置相应的Sugar属性更易读的语法。

在UML的类框中，方法显示在第三部分，如图8-4所示。

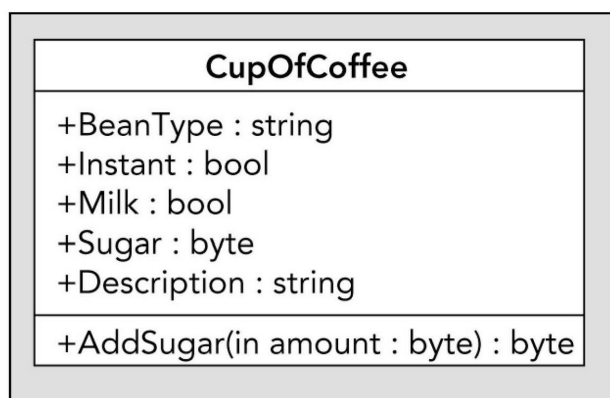


图8-4

其语法类似于字段和属性，但最后显示的类型是返回类型，在这一部分，还显示了方法的参数。在UML中，每个参数都带有下述标识符之一：**return**、**in**、**out**或**inout**。它们用于表示数据流的方向，其中**out**和

inout大致对应于第6章讨论的C#关键字out和ref。in大致对应于C#中不使用这两个关键字的情形（默认情形）。return表示传回调用方法的值。

8.1.2 一切皆对象

本书一直在使用对象、属性和方法。实际上，C#和.NET Framework中的所有东西都是对象。控制台应用程序中的Main()函数就是类的一个方法。前面介绍的每个变量类型都是一个类。前面使用的每个命令都是属性或方法，例如<String>.Length和<String>.ToUpper()等。句点字符把对象实例名与属性或方法名分隔开来，方法名后面的()把方法与属性区分开来。

对象无处不在，使用它们的语法通常比较简单，至少到现在为止都足够简单，使我们可以集中精力讨论C#中一些比较基础的方面。从现在开始详细介绍对象。这里讨论的概念都具有深远影响，它们甚至适用于简单的int变量。

8.1.3 对象的生命周期

每个对象都有一个明确定义的生命周期，除了“正在使用”的正常状态之外，还有两个重要的阶段：

- 构造阶段：第一次实例化一个对象时，需要初始化该对象。这个初始化过程称为构造阶段，由构造函数完成。
- 析构阶段：在删除一个对象时，常常需要执行一些清理工作，例如释放内存，这由析构函数完成。

1. 构造函数

对象的初始化过程是自动完成的。我们不需要自己寻找适于存储新对象的内存空间。但是，在初始化对象的过程中，有时需要执行一些额外工作。例如，需要初始化对象存储的数据。构造函数就是用于初始化数据的函数。

所有的类定义都至少包含一个构造函数。在这些构造函数中，可能有一个默认构造函数，该函数没有参数，与类同名。类定义还可能包含几个带有参数的构造函数，称为非默认的构造函数。代码可以使用它们以许多方式实例化对象，例如给存储在对象中的数据提供初始值。

在C#中，用new关键字来调用构造函数。例如，可用下面的方式通过其默认的构造函数实例化一个CupOfCoffee对象：

```
CupOfCoffee myCup = new CupOfCoffee();
```

还可以用非默认的构造函数来实例化对象。例如，CupOfCoffee类有一个非默认的构造函数，它使用一个参数在初始化时设置咖啡豆的品牌：

```
CupOfCoffee myCup = new CupOfCoffee("Blue Mountain");
```

构造函数与字段、属性和方法一样，可以是公共或私有的。在类外部的代码不能使用私有构造函数实例化对象，而必须使用公共构造函数。这样，通过把默认构造函数设置为私有的，就可以强制类的用户使用非默认的构造函数。

一些类没有公共的构造函数，外部的代码就不可能实例化它们，这

些类称为不可创建的类，但如稍后所述，这些类并不是完全没有用的。

2. 析构函数

.NET Framework使用析构函数来清理对象。一般情况下，不需要提供析构函数的代码，而由默认的析构函数自动执行操作。但是，如果在删除对象实例前需要完成一些重要操作，就应提供具体的析构函数。

例如，如果变量超出了范围，代码就不能访问它，但该变量仍存在于计算机内存的某个地方。只有在.NET运行程序执行其垃圾回收，进行清理时，该实例才被彻底删除。

8.1.4 静态成员和实例类成员

属性、方法和字段等成员是对象实例所特有的，此外，还有静态成员（也称为共享成员，尤其是Visual Basic用户常使用这个术语），例如静态方法、静态属性或静态字段。静态成员可以在类的实例之间共享，所以可以将它们看成类的全局对象。静态属性和静态字段可以访问独立于任何对象实例的数据，静态方法可以执行与对象类型相关但与对象实例无关的命令。在使用静态成员时，甚至不需要实例化对象。

例如，前面使用的`Console.WriteLine()`和`Convert.ToString()`方法就是静态的，根本不需要实例化`Console`或`Convert`类（如果试着进行这样的实例化，操作会失败，因为这些类的构造函数不是可公共访问的，如前所述）。

许多情况下，静态属性和静态方法有很好的效果。例如，可以使用

静态属性跟踪给类创建了多少个实例。在UML语法中，类的静态成员带有下划线，如图8-5所示。

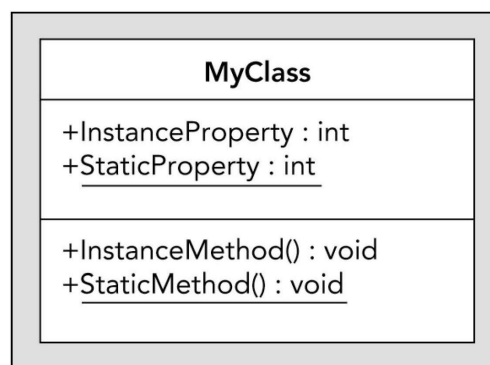


图8-5

使用类中的静态成员时，需要预先初始化这些成员。在声明时，可以给静态成员提供一个初始值，但有时需要执行更复杂的初始化操作，或者在赋值、执行静态方法之前执行某些操作。

使用静态构造函数可以执行此类初始化任务。一个类只能有一个静态构造函数，该构造函数不能有访问修饰符，也不能带任何参数。静态构造函数不能直接调用，只能在下述情况下执行：

- 创建包含静态构造函数的类实例时
- 访问包含静态构造函数的类的静态成员时

在这两种情况下，会首先调用静态构造函数，之后实例化类或访问静态成员。无论创建了多少个类实例，其静态构造函数都只调用一次。为了区分静态构造函数和本章前面介绍的构造函数，也将所有非静态构造函数称为实例构造函数。

2. 静态类

我们常常希望类只包含静态成员，且不能用于实例化对象（如 Console）。为此，一种简单的方法是使用静态类，而不是把类的构造

函数设置为私有。静态类只能包含静态成员，不能包含实例构造函数，因为按照定义，它根本不能实例化。但静态类可以有一个静态构造函数，如上一节所述。

注意： 如果以前完全没有接触过OOP，在阅读本章的其他内容之前，应该停下来将OOP研究一番。在学习更复杂的OOP内容之前，全面掌握基础知识是很重要的。

8.2 OOP技术

前面介绍了一些基础知识，知道对象是什么，以及对象的工作原理，下面讨论对象的其他一些特性，包括：

- 接口
- 继承
- 多态性
- 对象之间的关系
- 运算符重载
- 事件
- 引用类型和值类型

8.2.1 接口

接口是把公共实例（非静态）方法和属性组合起来，以封装特定功能的一个集合。一旦定义了接口，就可以在类中实现它。这样，类就可以支持接口所指定的所有属性和成员。

注意，接口不能单独存在。不能像实例化一个类那样实例化接口。另外，接口不能包含实现其成员的任何代码，而只能定义成员本身。实现过程必须在实现接口的类中完成。

在前面的咖啡示例中，可以把通用属性和方法，例如AddSugar()、Milk、Sugar和Instant组合到一个接口中，这个接口可以命名为

IHotDrink（接口名一般以大写字母I开头）。然后就可以在其他对象上使用该接口，例如CupOfTea类的对象。所以可以采用类似方式处理这些对象，而对象仍保有自己的属性（例如CupOfCoffee仍有属性BeanType，CupOfTea仍有属性LeafType）。

在UML中，在对象上实现的接口用“棒棒糖”语法来表示。在图8-6中，用与类相似的语法把IHotDrink的成员放在一个单独的框中。

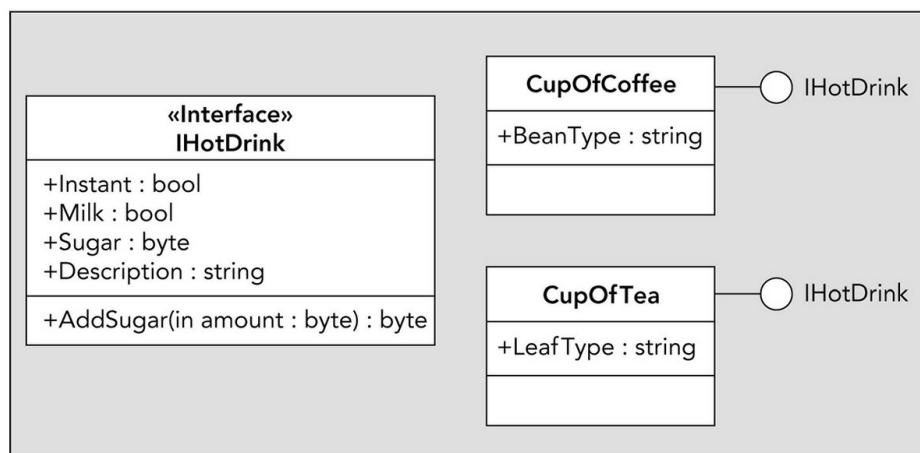


图8-6

一个类可以支持多个接口，多个类也可以支持相同的接口。所以接口的概念让用户和其他开发人员更容易理解其他人的代码。例如，有一些代码使用一个带某接口的对象。假定不使用这个对象的其他属性和方法，就可以用另一个对象代替这个对象（例如，使用上述IHotDrink接口的代码可以处理CupOfCoffee和CupOfTea实例）。另外，该对象的开发人员可以提供该对象的更新版本，只要它支持已经在用的接口，就可以在代码中使用这个新版本。

发布接口后，即接口可以用于其他开发人员或终端用户后，最好不要修改它。理解这一点的一种方式是把接口看成类的创建者和使用者之

间的协定，即“每个支持接口X的类都支持这些方法和属性”。如果以后修改了接口，也许是升级了底层的代码，该接口的使用者就不能正确运行接口，甚至失败。所以，我们应做的是创建一个新接口，使其扩展旧接口，可能还包含一个版本号，如X2。这是创建接口的标准方式，以后我们会常常遇到已编号的接口。

可删除的对象

IDisposable接口特别有趣。支持IDisposable接口的对象必须实现Dispose()方法，即它们必须提供这个方法的代码。当不再需要某个对象（例如，在对象超出作用域之前）时，就调用这个方法，释放重要资源，否则，等到对垃圾回收调用析构方法时才会释放该资源。这样可以更好地控制对象所用的资源。

C#允许使用一种可以优化使用这个方法的结构。using关键字可以在代码块中初始化使用重要资源的对象，在这个代码块的末尾会自动调用Dispose()方法，用法如下：

```
<ClassName  
  
  
><VariableName  
  
  
> = new<ClassName
```

```
>();
    ...
    using (<VariableName
```

```
>)
{
    ...
}
```

或者把初始化对象<VariableName>作为using语句的一部分：

```
using (<ClassName
```

```
><VariableName
```

```
> = new<ClassName
```

```
>())
{
    ...
}
```

这两种情况下，可在using代码块中使用变量<VariableName>，并在代码块的末尾自动删除（在代码块执行完毕后，调用Dispose()）。

8.2.2 继承

继承是OOP最重要的特性之一。任何类都可以从另一个类继承，这就是说，这个类拥有它继承的类的所有成员。在OOP中，被继承（也称为派生）的类称为父类（也称为基类）。注意，C#中的对象仅能直接派生于一个基类，当然基类也可以有自己的基类。

继承性可从一个较一般的基类扩展或创建更多的特定类。例如，考虑一个代表农场家畜的类（由80多岁的资深开发人员MacDonald在他的家畜应用程序中使用）。这个类名为Animal，拥有EatFood()或Breed()等方法，我们可以创建一个派生类Cow；Cow支持所有这些方法，也有自己的方法，如Moo()和SupplyMilk()。还可以创建另一个派生类Chicken，该类有Cluck()和LayEgg()方法。

在UML中，用箭头表示继承，如图8-7所示。

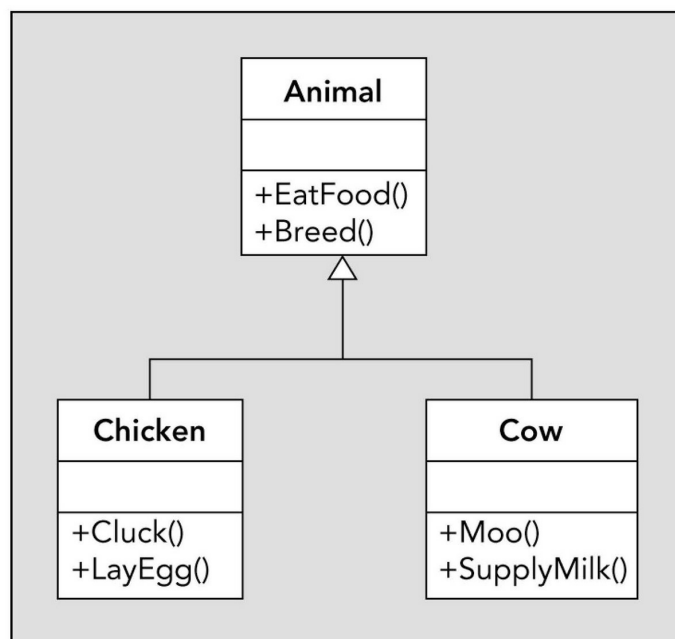


图8-7

注意：为简洁起见，图8-7中省略了成员的返回类型。

在继承一个基类时，成员的可访问性就成了一个重要问题。派生类不能访问基类的私有成员，但可以访问其公共成员。不过，派生类和外部的代码都可以访问公共成员。这就是说，只使用这两个级别的可访问性，不能让一个成员可由基类和派生类访问，而不能由外部的代码访问。

为解决这个问题，C#提供了第三种可访问性：`protected`，只有派生类才能访问`protected`成员。对于外部代码来说，这个可访问性与私有成员一样：外部代码不能访问`private`成员和`protected`成员。

除了定义成员的保护级别外，我们还可以为成员定义其继承行为。基类的成员可以是虚拟的，也就是说，成员可以由继承它的类重写。派生类可以提供成员的另一种实现代码。这种实现代码不会删除原来的代码，仍可以在类中访问原来的代码，但外部代码不能访问它们。如果没有提供其他实现方式，通过派生类使用成员的外部代码就自动访问基类中成员的实现代码。

注意：虚拟成员不能是私有成员，因为这样会自相矛盾——不能既要求派生类重写成员，又不让派生类访问该成员。

在前面的家畜示例中，可以把EatFood()变成虚拟成员，在派生类中为它提供新的实现代码，例如为Cow类提供新的实现代码，如图8-8所示。这里显示了Animal和Cow类的EatFood()方法，说明它们有自己的实现代码。

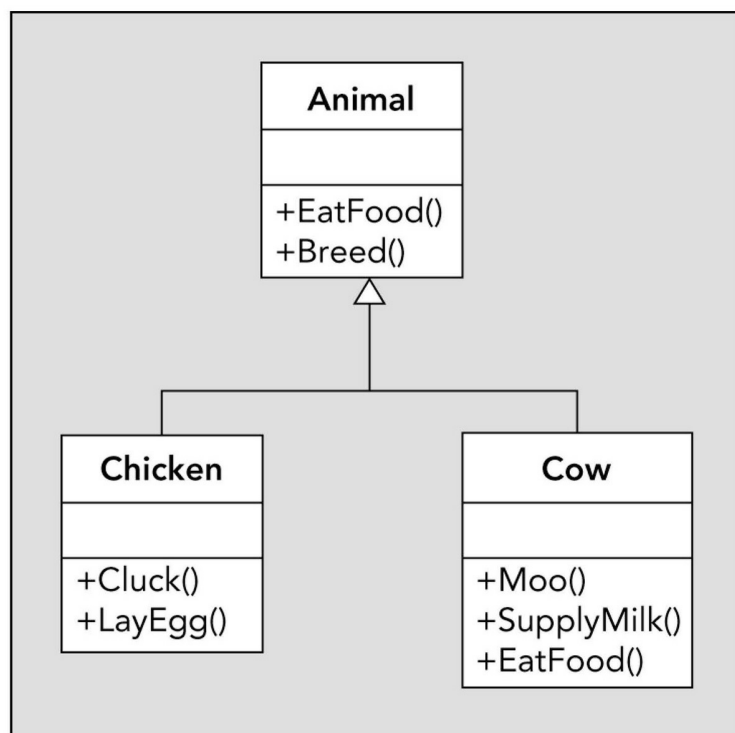


图8-8

基类还可以定义为抽象类。抽象类不能直接实例化。要使用抽象类，必须继承这个类，抽象类可以有抽象成员，这些成员在基类中没有实现代码，所以派生类必须实现它们。如果Animal是一个抽象类，UML就会如图8-9所示。

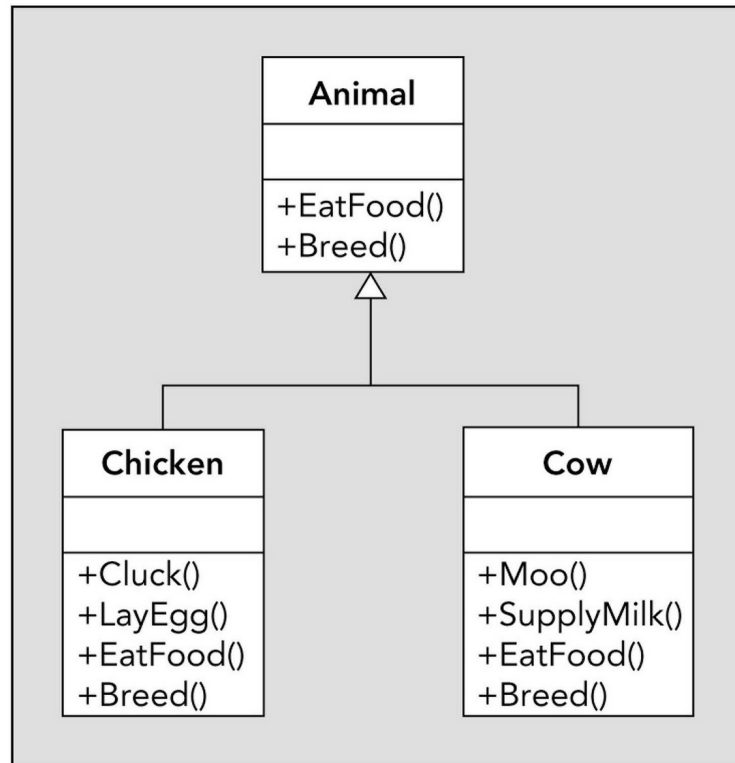


图8-9

注意： 抽象类名以斜体显示（有时它们的方框有一个短横线）。

在图8-9中，`EatFood()`和`Breed()`都显示在派生类`Chicken`和`Cow`中，这说明这些方法是抽象的（必须在派生类中重写）或者虚拟的（这里已经在`Chicken`和`Cow`中重写）。当然，抽象基类可以提供成员的实现代码，这是十分常见的。不能实例化抽象类，并不意味着不能在抽象类中封装功能。

最后，类可以是密封（`seal`）的。密封的类不能用作基类，所以没有派生类。

在C#中，所有对象都有一个共同的基类object（在.NET Framework中，它是System.Object类的别名）。第9章将详细介绍这个类。

注意：如本章前面所述，接口也可以继承自其他接口。与类不同的是，接口可以继承多个基接口（与类可以支持多个接口的方式类似）。

8.2.3 多态性

继承的一个结果是派生于基类的类在方法和属性上有一定的重叠，因此，可以使用相同的语法处理从同一个基类实例化的对象。例如，如果基类Animal有一个EatFood()方法，则在其派生类Cow和Chicken中调用这个方法语法是类似的：

```
Cow myCow = new Cow();  
Chicken myChicken = new Chicken();  
myCow.EatFood();  
myChicken.EatFood();
```

多态性则更推进了一步。可以把某个派生类型的变量赋给基本类型的变量，例如：

```
Animal myAnimal = myCow;
```

不需要进行强制类型转换，就可以通过这个变量调用基类的方法：

```
myAnimal.EatFood();
```

结果是调用派生类中的EatFood()的实现代码。注意，不能以相同的方式调用派生类上定义的方法。下面的代码无法运行：

```
myAnimal.Moo();
```

但可以把基本类型的变量转换为派生类变量，调用派生类的方法，如下所示：

```
Cow myNewCow = (Cow)myAnimal;  
myNewCow.Moo();
```

如果原始变量的类型不是Cow或派生于Cow的类型，这个强制类型转换就会引发一个异常。有许多方式说明对象的类型是什么，详见下一章。

在派生于同一个类的不同对象上执行任务时，多态性是一种极有效的技巧，其使用的代码最少。注意并不是只有共享同一个父类的类才能利用多态性。只要子类 and 孙子类在继承层次结构中有一个相同的类，它们就可以用同样的方式利用多态性。

还要注意，在C#中，所有类都派生于同一个类object，object是继承层次结构中的根。所以可以把所有对象看成object类的实例。这就是在建立字符串时，WriteLine()可以处理无数多种参数组合的原因。第一个参数后面的每个参数都可以看成一个object实例，所以可以把任何对象的输出结果写到屏幕上。为此，需要调用方法ToString()（object的一个成员）。我们可以重写这个方法，为自己的类提供合适的实现代码，或者使用默认实现代码，返回类名（根据它所在的名称空间，返回类的限

定名称）。

接口的多态性

尽管不能像对象那样实例化接口，但可以建立接口类型的变量，然后就可以在支持该接口的对象上，使用这个变量来访问该接口提供的方法和属性。

例如，假定不使用基类Animal提供的EatFood()方法，而是把该方法放在IConsume接口上。Cow和Chicken类也支持这个接口，唯一的区别是它们必须提供EatFood()方法的实现代码（因为接口不包含实现代码），接着就可以使用下述代码访问该方法了：

```
Cow myCow = new Cow();  
Chicken myChicken = new Chicken();  
IConsume consumeInterface;
```

```
consumeInterface = myCow;
```

```
consumeInterface.EatFood();
```

```
consumeInterface = myChicken;
```

```
consumeInterface.EatFood();
```

这就提供了以相同方式访问多个对象的简单方式，且不依赖于一个公共的基类。例如，这个接口可以由派生于Vegetable而不是Animal的VenusFlyTrap类实现：

```
VenusFlyTrap myVenusFlyTrap = new VenusFlyTrap();  
IConsume consumeInterface;  
consumeInterface = myVenusFlyTrap;  
consumeInterface.EatFood();
```

在这段代码中，调用consumeInterface.EatFood()的结果是调用Cow、Chicken或VenusFlyTrap类的EatFood()方法，这取决于把哪个实例赋予接口类型的变量。

注意，派生类会继承其基类支持的接口。在上面的第一个示例中，要么是Animal支持IConsume，要么是Cow和Chicken支持IConsume。有共同基类的类不一定有共同接口，反之亦然。

8.2.4 对象之间的关系

继承是对象之间的一种简单关系，可以让派生类完整地获得基类的特性，而且派生类也可以访问基类内部的一些工作代码（通过受保护的成员）。对象之间还具有其他一些重要关系。

本节简要讨论下述关系：

- 包含关系： 一个类包含另一个类。这类似于继承关系，但包含类可以控制对被包含类的成员的访问，甚至在使用被包含类的成员前进行其他处理。
- 集合关系： 一个类用作另一个类的多个实例的容器。这类似于对象数组，但集合具有其他功能，包括索引、排序和重新设置大小等。

1. 包含关系

用一个成员字段包含对象实例，就可以实现包含（containment）关系。这个成员字段可以是公共字段，此时与继承关系一样，容器对象的用户就可以访问它的方法和属性，但不能像继承关系那样，通过派生类访问类的内部代码。

另外，可以让被包含的成员对象变成私有成员。如果这么做，用户就不能直接访问任何成员，即使这些成员是公共的。但可以使用包含类的成员访问这些私有成员。也就是说，可以完全控制被包含的类对外提供什么成员（或者不提供任何成员），还可以在访问被包含类的成员前，在包含类的成员上执行其他处理。

例如，Cow类包含一个Udder类，Udder类有一个公共方法Milk()。Cow对象可以按照要求调用这个方法，作为其SupplyMilk()方法的一部

分，但Cow对象的用户看不到这些细节，或者这些细节对Cow对象的用户并不重要。

在UML中，被包含类可以用关联线条来表示。对于简单包含关系，可以用带有1的线条说明一对一的关系（一个Cow实例包含一个Udder实例）。为清晰起见，也可以把被包含的Udder类实例表示为Cow类的私有字段，如图8-10所示。

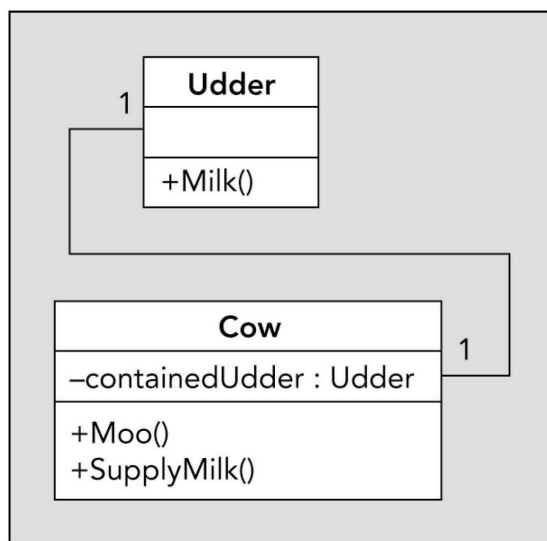


图8-10

2. 集合关系

第5章讨论了如何使用数组存储多个同类型变量，这也适用于对象（前面使用的变量类型实际上是对象）。例如：

```
Animal[] animals = new Animal[5];
```

集合基本上就是一个增加了功能的数组。集合以与其他对象相同的方式实现为类。它们通常以所存储的对象名称的复数形式来命名，例如用类**Animals**包含**Animal**对象的一个集合。

数组与集合的主要区别是，集合通常实现额外的功能，例如**Add()**和**Remove()**方法可添加和删除集合中的项。而且集合通常有一个**Item**属性，它根据对象的索引返回该对象。通常，这个属性还允许实现更复杂的访问方式。例如，可以设计一个**Animals**，让**Animal**对象根据其名称

来访问。

其UML表示如图8-11所示。图8-11中没有包含成员，因为这里描述的是关系。连接线末尾的数字表示一个Animals对象可以包含0个或多个Animal对象。第11章将详细论述集合。

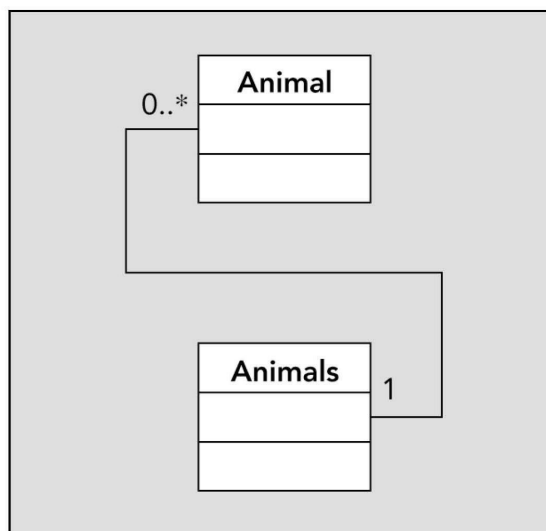


图8-11

8.2.5 运算符重载

本书前面介绍了如何使用运算符处理简单的变量类型。有时也可以把运算符用于从类实例化而来的对象，因为类可以包含如何处理运算符的指令。

例如，给Animal添加一个新属性Weight。接着使用下述代码比较家畜的体重：

```
if (cowA.Weight > cowB.Weight)
```

```
{
    ...
}
```

使用运算符重载，可在代码中提供隐式使用Weight属性的逻辑，如

下面的代码所示：

```
if (cowA > cowB)
```

```
{  
    ...  
}
```

大于运算符>被重载了。我们为重载运算符编写代码，执行上述操作，这段代码用作类定义的一部分，而该运算符作用于这个类。在上面的示例中，使用了两个Cow对象，所以运算符重载定义包含在Cow类中。也可以采用相同的方式重载运算符，使其处理不同的类，其中一个（或两个）类定义包含达到这一目的的代码。

注意，只能采用这种方式重载现有的C#运算符，不能创建新的运算符。但可以为一元和二元运算符（如+或>）提供实现代码。详见第13章。

8.2.6 事件

对象可以激活和使用事件，作为它们处理的一部分。事件是非常重要的，可以在代码的其他部分起作用，类似于异常（但功能更强大）。例如，可以在把Animal对象添加到Animals集合中时，执行特定的代码，而这部分代码不是Animals类的一部分，也不是调用Add()方法的代码的一部分。为此，需要给代码添加事件处理程序，这是一种特殊类型

的函数，在事件发生时调用。还需要配置这个处理程序，以监听自己感兴趣的事件。

使用事件可以创建事件驱动的应用程序，此类应用程序比读者此时所能想到的多得多。例如，许多Windows应用程序完全依赖于事件。每个按钮单击或滚动条拖动操作都是通过事件处理实现的，其中事件是通过鼠标或键盘触发的。

本章后面将介绍Windows应用程序中事件的工作原理，第13章将深入讨论事件。

8.2.7 引用类型和值类型

在C#中，数据根据变量的类型以两种方式中的一种存储在一个变量中。变量的类型分为两种：引用类型和值类型，其区别如下：

- 值类型在内存的同一处存储它们自己和它们的内容。
- 引用类型存储指向内存中其他某个位置（称为堆）的引用，实际内容存储在这个位置。

实际上，在使用C#时，不必过多地考虑这个问题。到目前为止，所使用的string变量（这是引用类型）与使用其他简单变量（大多数是值类型，例如int）的方式完全相同。

值类型和引用类型的一个主要区别是：值类型总是包含一个值，而引用类型可以是null，表示它们不包含值。但是，可使用可空类型创建值类型，使值类型在这个方面的行为方式类似于引用类型（即可以为null）。第12章在介绍泛型（包括可空类型）这一高级主题时将讨论这

方面的内容。

只有string和object类型是简单的引用类型。数组也是隐式的引用类型。我们创建的每个类都是引用类型，这就是在这里说明这一点的原因。

8.3 桌面应用程序中的OOP

第2章在C#中使用Windows Presentation Foundation (WPF) 创建了一个简单的桌面应用程序。WPF桌面应用程序非常依赖OOP技术，本节将论述OOP技术，说明本章的一些论点。下面通过一个简单示例加以说明。

试一试：使用对象：**Ch08Ex01**

(1) 在C:\BegVCSharp\Chapter08目录中创建一个新的WPF应用程序Ch08Ex01。

(2) 使用Toolbox添加一个新的按钮控件，使其位于MainWindow的中央，如图8-12所示。

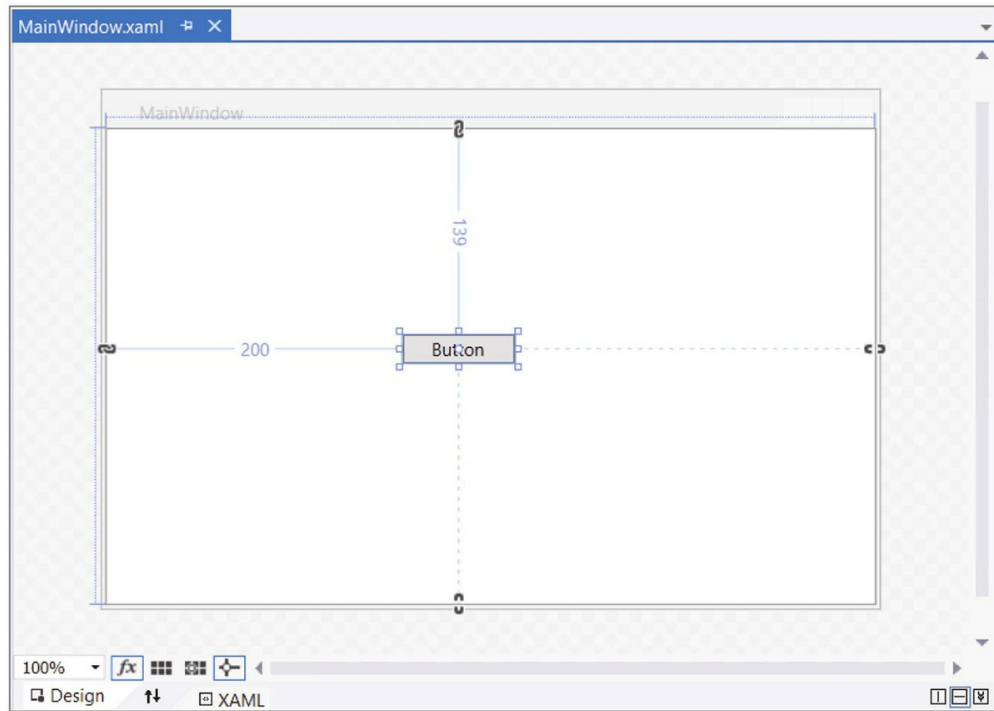


图8-12

(3) 双击按钮，为鼠标单击事件添加代码。修改代码，如下所示：

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
```

```
    ((Button)sender).Content = "Clicked!";
```

```
    Button newButton = new Button();
```



```
newButton.Content = "New Button! ";
```

```
newButton.Margin = new Thickness(10, 10, 200, 200);
```

```
newButton.Click += newButton_Click;
```

```
((Grid)((Button)sender).Parent).Children.Add(newButton);
```

```
}
```

```
private void newButton_Click(object sender, RoutedEventArgs e)
```

```
{
```

```
((Button)sender).Content = "Clicked!!";
```

```
}
```

(4) 运行应用程序，窗口如图8-13所示。

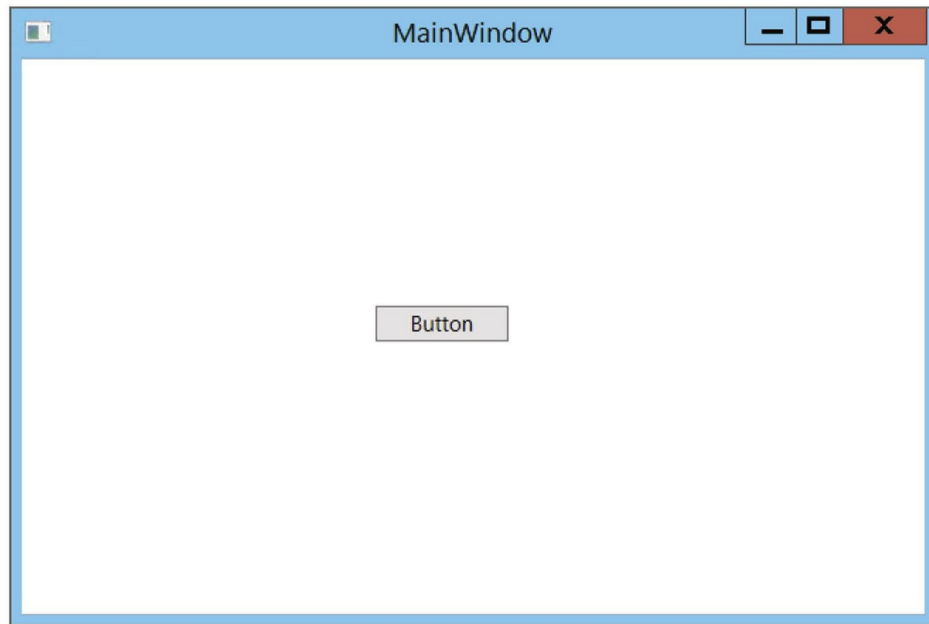


图8-13

(5) 单击标记为Button的按钮，显示内容将随之变化，如图8-14所示。

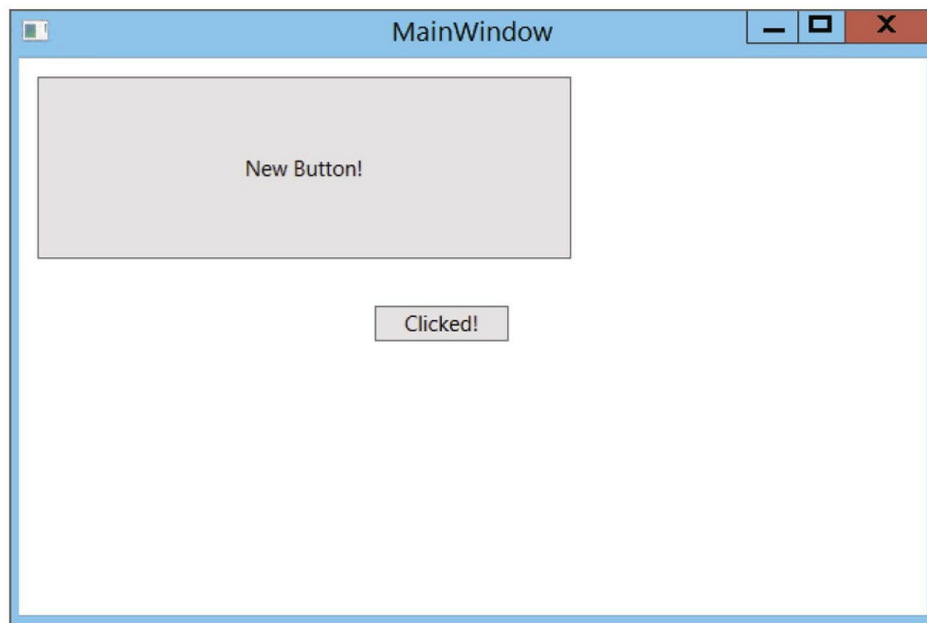


图8-14

（6）单击标记为New Button!的按钮，显示内容将随之变化，如图8-15所示。

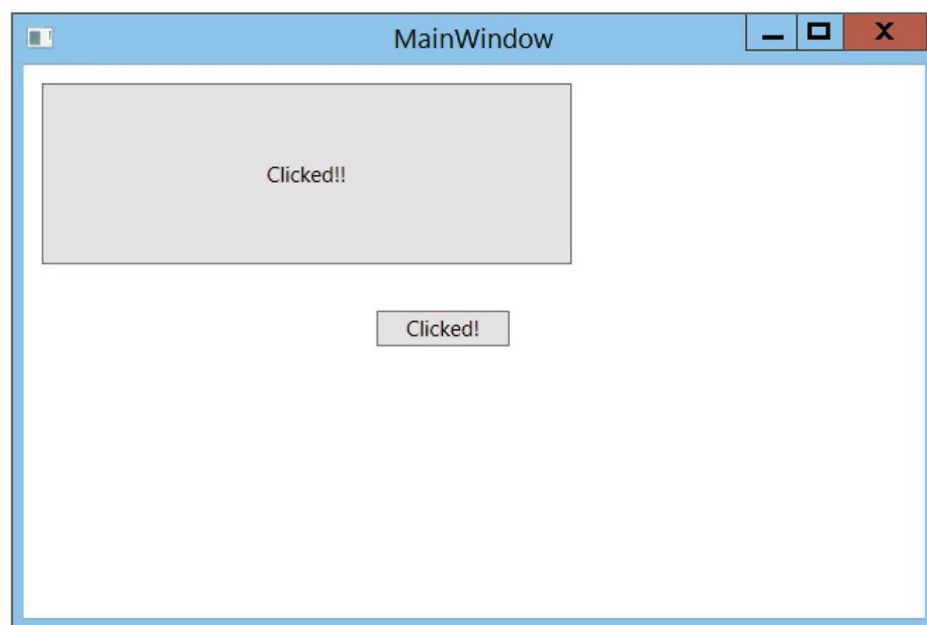


图8-15

示例说明

仅添加几行代码，就创建了一个可以完成某项任务的桌面应用程序。下面说明C#中的一些OOP技术。即使在谈到桌面应用程序时，“一切皆对象”这句话也是正确的。从运行的窗体，到窗体上的控件，都需要使用OOP技术。这个示例重点说明本章前面介绍的一些概念，解释如何把它们组合在一起。

在应用程序中，首先在MainWindow窗口中添加一个新按钮，这个按钮是一个对象，它是Button类的一个实例；窗口是MainWindow类的实例，该类从Window类派生而来。接着双击按钮，添加一个事件处理程序，监听Button类提供的Click事件。这个事件处理程序被添加到封装应用程序的MainWindow对象代码中，是一个私有方法：

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
}
```

这段代码使用C#关键字private作为修饰符。现在不要考虑这个关键字，第9章将详细解释本章提及的OOP技术。

我们添加的第一行代码改变了所单击按钮上的文本。它利用了本章前面讨论的多态性。表示按钮的Button对象作为一个object参数发送给事件处理程序，该事件处理程序把参数强制转换为Button类型（这是可能的，因为Button对象继承于System.Object，System.Object是一个.NET类，object是其别名）。然后修改对象的Content属性，改变显示的文本：

```
((Button)sender).Content = "Clicked!";
```

接着用new关键字创建一个新的Button对象（注意在这个项目中设置了名称空间，因此可以使用这个简单的语法，否则就需要使用这个对象的完整限定名System.Windows.Forms.Button）：

```
Button newButton = new Button();
```

还可以将新建的Button对象的Content和Margin属性设置为合适的值，使按钮显示在合适的地方。注意，Margin属性的类型是Thickness，因此使用非默认构造函数创建一个Thickness对象，然后将其赋值给Margin属性：

```
newButton.Content = "New Button!";  
newButton.Margin = new Thickness(10, 10, 200, 200);
```

在代码的其他地方添加一个新的事件处理程序，以响应新按钮生成的Click事件：

```
private void newButton_Click(object sender, RoutedEventArgs e)  
{  
    ((Button)sender).Content = "Clicked!!";  
}
```

接着使用重载运算符语法，把这个事件处理程序注册为Click事件的监听程序：

```
newButton.Click += newButton_Click;
```

最后，把新按钮添加到窗口中。为此，使用已有按钮的Parent属性找出其父对象，将其转换为正确类型，即Grid。然后，通过将新按钮作为参数传递给Grid.Children属性的Add()方法，将该按钮添加到窗口中：

```
((Grid)((Button)sender).Parent).Children.Add(newButton);
```

这些代码实际上没有看起来那样复杂。一旦理解了WPF是通过一个控件（包括按钮和容器）的层次结构来显示窗口的内容，使用这类代码就显得再自然不过。

这个简短示例几乎使用了本章介绍的所有技术。可以看出，OOP编程并不复杂——只需要从另一个角度来看待编程即可。

8.4 练习

(1) 下述哪些项在OOP中有真实级别的可访问性？

- 友元
- 公共
- 安全
- 私有
- 受保护的
- 松散的
- 通配符

(2) “必须手动调用对象的析构函数，否则就会浪费内存”的说法正确吗？

(3) 在调用类的静态方法时，需要创建该类的对象吗？

(4) 为下述类和接口绘制一个类似于本章介绍的图形的UML图：

- 抽象类HotDrink，它具有方法Drink、AddMilk和AddSugar，以及属性Milk和Sugar。
- 接口ICup，它具有方法Refill和Wash，以及属性Color和Volume。
- 派生于HotDrink的类CupOfCoffee支持ICup接口，还有一个属性BeanType。
- 派生于HotDrink的类CupOfTea支持ICup接口，还有一个属性LeafType。

(5) 为一个函数编写一些代码，接受上述示例的两个杯子对象中的任意一个作为参数。该函数应该为传递给它的任何杯子对象调用AddMilk、Drink和Wash方法。

附录A给出了练习答案。

8.5 本章要点

主题	要点
对象和类	对象是OOP应用程序的组成部件。类是用于实例化对象的类型定义。对象可以包含数据，提供其他代码可以使用的操作。数据可以通过属性供外部代码使用，操作可以通过方法供外部代码使用。属性和方法都称为类的成员。属性可以进行读取访问、写入访问或读写访问。类成员可以是公共的（可用于所有代码）或私有的（只有类定义中的代码可以使用）。在.NET中，所有的东西都是对象
对象的生命周期	对象通过调用它的一个构造函数来实例化。不再需要对象时，就执行其析构函数，以删除它。要清理对象，常常需要手工删除它
静态成员和实例成员	实例成员只能在类的对象实例上使用，静态成员只能直接通过类定义使用，它不与实例关联
接口	接口是可以在类上实现的公共属性和方法的集合。可将实现了一个接口的类的对象赋值给对应实例类型的变量。之后通过该变量，可以使用该接口定义的成员
继承	继承是一个类定义派生于另一个类定义的机制。类从其父类中继承成员，每个类都只能有一个父类。子类不能访问父类的私有成员，但可以定义受保护的成员，受保护的成员只能在该类和派生于该类的子类中使用。子类可以重写父类中的虚拟成员。所有的类都有一个以System.Object结尾的继承链，在C#中，System.Object有一个别名object
多态性	从一个派生类实例化的所有对象都可以看成其父类的实例

对象关系和特性

对象可以包含其他对象，也可以表示其他对象的集合。要在表达式中处理对象，常常需要通过运算符重载，定义运算符如何处理对象。对象可以提供事件，事件因某种内部处理而被触发，客户端代码通过提供事件处理程序来响应事件

第9章 定义类

本章内容：

- 如何在C#中定义类和接口
- 用来控制可访问性和继承的关键字的使用
- `System.Object`类及其在类定义中的作用
- 如何使用VS提供的一些帮助工具
- 如何定义类库
- 接口和抽象类的异同
- 结构类型的更多内容
- 复制对象的一些重要信息

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 9 Code后，可以找到与本章示例对应的单独文件。

第8章介绍了面向对象编程（OOP）的特性，本章将理论付诸实践，看看如何在C#中定义类。本章并不讨论如何定义类的成员，而重点讨论如何定义类本身。

首先分析基本的类定义语法、用于确定类可访问性的关键字以及指定继承的方式。我们还将介绍接口的定义，因为它们在许多方面都类似于类的定义。

本章的其他部分介绍在C#中定义类时涉及的其他主题。

9.1 C#中的类定义

C#使用class关键字来定义类：

class

```
MyClass
{
    // Class members.
}
```

这段代码定义了一个类MyClass。定义了一个类后，就可以在项目中能访问该定义的其他位置对该类进行实例化。默认情况下，类声明为内部的，即只有当前项目中的代码才能访问它。可使用internal访问修饰符关键字来显式地指定这一点，如下所示（但这没有必要）：

internal

```
class MyClass
{
    // Class members.
}
```

另外，还可以指定类是公共的，可由其他项目中的代码来访问。为

此，要使用关键字**public**:

public

```
class MyClass
{
    // Class members.
}
```

除了这两个访问修饰符关键字外，还可以指定类是抽象的（不能实例化，只能继承，可以有抽象成员）或密封的（**sealed**，不能继承）。为此，可使用两个互斥的关键字**abstract**或**sealed**。所以，必须使用下述方式声明抽象类：

public abstract

```
class MyClass
{
    // Class members, may be abstract.
}
```

其中**MyClass**是一个公共抽象类，也可以是内部抽象类。

密封类的声明如下所示：

public sealed

```
class MyClass
{
    // Class members.
}
```

与抽象类一样，密封类也可以是公共的或内部的。

还可以在类定义中指定继承。为此，要在类名的后面加上一个冒号，其后是基类名，例如：

```
public class MyClass : MyBase
```

```
{
    // Class members.
}
```

注意，在C#的类定义中，只能有一个基类。如果继承了一个抽象类，就必须实现所继承的所有抽象成员（除非派生类也是抽象的）。

编译器不允许派生类的可访问性高于基类。也就是说，内部类可以继承于一个公共基类，但公共类不能继承于一个内部基类。因此，下述代码是合法的：

```
public
```

```
class MyBase
{
    // Class members.
}
internal
```

```
class MyClass : MyBase
{
    // Class members.
}
```

但下述代码不能编译:

```
internal
```

```
class MyBase
{
    // Class members.
}
public
```

```
class MyClass : MyBase
{
    // Class members.
```



```
}
```

如果没有使用基类，被定义的类就只继承于基类`System.Object`（它在C#中的别名是`object`）。毕竟，在继承层次结构中，所有类的根都是`System.Object`，稍后将详细介绍这个基类。

除了以这种方式指定基类外，还可在冒号之后指定支持的接口。如果指定了基类，它必须紧跟在冒号的后面，之后才是指定的接口。如果未指定基类，接口就跟在冒号的后面。必须使用逗号来分隔基类名（如果有基类的话）和接口名。

例如，给`MyClass`添加一个接口，如下所示：

```
public class MyClass : IMyInterface

{
    // Class members.
}
```

支持该接口的类必须实现所有接口成员，但如果不想使用给定的接口成员，可以提供一种“空”的实现方式（没有函数代码）。还可以把接口成员实现为抽象类中的抽象成员。

下面的声明是无效的，因为基类`MyBase`不是继承列表中的第一项：

```
public class MyClass : IMyInterface, MyBase
```

```
{  
    // Class members.  
}
```

指定基类和接口的正确方式如下：

```
public class MyClass : MyBase, IMyInterface
```

```
{  
    // Class members.  
}
```

可以指定多个接口，所以下列代码也是有效的：

```
public class MyClass : MyBase, IMyInterface, IMySecondInterfac
```

```
{  
    // Class members.  
}
```

表9-1列出了类定义中可以使用的访问修饰符的组合。

表9-1 类定义中可以使用的访问修饰符

修饰符	含义
无或internal	只能在当前项目中访问类
public	可以在任何地方访问类
abstract或internal abstract	类只能在当前项目中访问，不能实例化，只能被继承
public abstract	类可以在任何地方访问，不能实例化，只能被继承
sealed或internal sealed	类只能在当前项目中访问，不能被继承，只能实例化
public sealed	类可以在任何地方访问，不能被继承，只能实例化

接口的定义

声明接口的方式与声明类的方式相似，但使用的关键字是interface而不是class，例如：

interface

IMyInterface

```
{
    // Interface members.
}
```

访问修饰符关键字`public`和`internal`的使用方式是相同的，与类一样，接口也默认定义为内部接口。所以要使接口可以公开访问，必须使用`public`关键字：

`public`

```
interface IMyInterface
{
    // Interface members.
}
```

不能在接口中使用关键字`abstract`和`sealed`，因为这两个修饰符在接口定义中是没有意义的（它们不包含实现代码，所以不能直接实例化，且必须是可继承的）。

也可以用与类继承类似的方式来指定接口的继承。主要的区别是可以使用多个基接口，例如：

```
public interface IMyInterface : IMyBaseInterface, IMyBaseInter

{
    // Interface members.
}
```

接口不是类，所以没有继承`System.Object`。但为了方便起见，

`System.Object`的成员可以通过接口类型的变量来访问。如上所述，不能用实例化类的方式来实例化接口。下面的示例提供了一些类定义的代码和使用它们的代码。

试一试：定义类：**Ch09Ex01\Program.cs**

(1) 在C:\BegVCSharp\Chapter09目录中创建一个新的控制台应用程序Ch09Ex01。

(2) 修改Program.cs中的代码，如下所示：

```
using static System.Console;
namespace Ch09Ex01
{
    public abstract class MyBase {}

    internal class MyClass : MyBase {}

    public interface IMyBaseInterface {}
```

```
internal interface IMyBaseInterface2 {}
```

```
internal interface IMyInterface : IMyBaseInterface, IMyBas
```

```
internal sealed class MyComplexClass : MyClass, IMyInterfa
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        MyComplexClass myObj = new MyComplexClass();
```

```
        WriteLine(myObj.ToString());
```

```
        ReadKey();
```

```
}  
}  
}
```

(3) 执行项目，结果如图9-1所示。

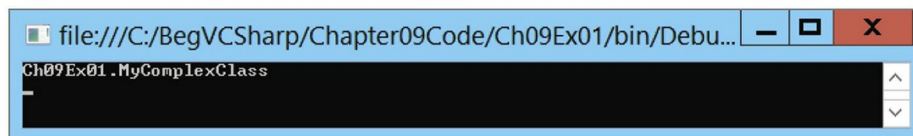


图9-1

示例说明

这个项目在下面的继承层次结构中定义了类和接口，如图9-2所示。

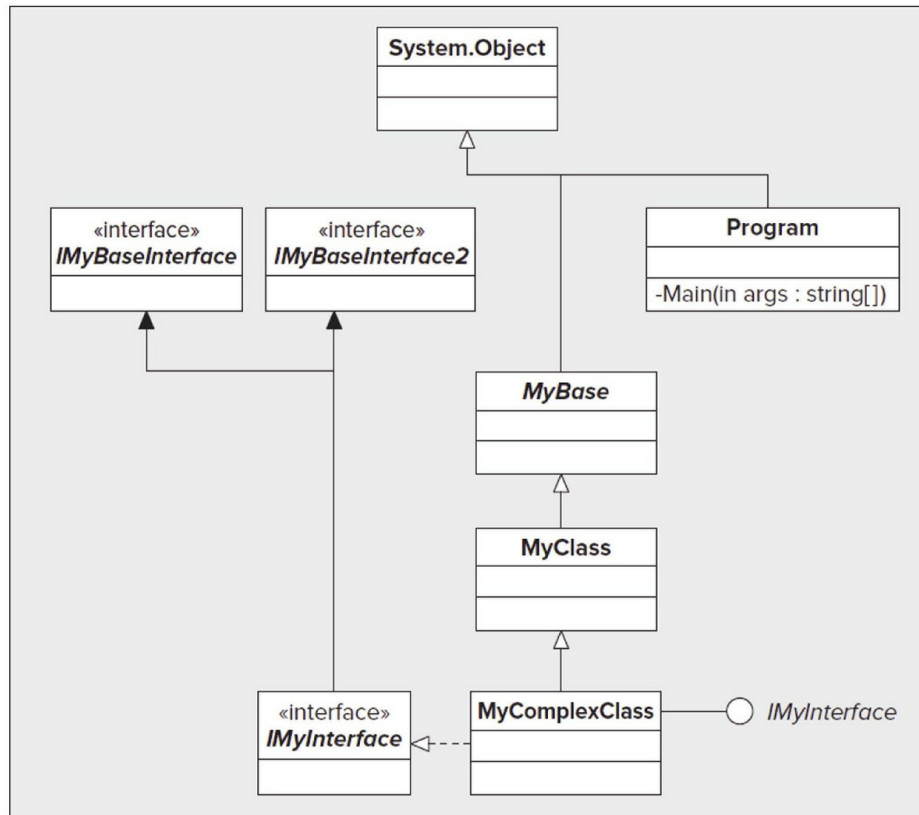


图9-2

这里之所以包含Program，是因为尽管这个类不是主要类层次结构的一部分，但是它的定义方式与其他类的定义方式相同。这个类处理的Main()方法是应用程序的入口点。

MyBase和IMyBaseInterface被定义为公共的，所以可以在其他项目中使用它们。其他类和接口都是内部的，只能在本项目中使用。

Main()中的代码调用MyComplexClass的一个实例myObj的ToString()方法：

```
MyComplexClass myObj = new MyComplexClass();
WriteLine(myObj.ToString());
```


这是继承自`System.Object`的一个方法（图中没有显示，为清晰起见，该图省略了这个类的成员），并把对象的类名作为一个字符串返回，该类名用相关的名称空间来限定。

这个示例没有完成什么具体工作，但本章后面还要利用这个示例演示几个重要概念和技术。

9.2 System.Object

因为所有类都继承于System.Object，所以这些类都可以访问该类中受保护的成员和公共成员。下面看看可供使用的成员有哪些。

System.Object包含的方法如表9-2所示。

表9-2 System.Object类的方法

方 法	返回类型	虚拟	静态	说 明
Object()	N/A	否	否	System.Object 类型的构造函数，由派生类型的构造函数自动调用
~Object()(也称为 Finalize(), 参见下一节)	N/A	否	否	System.Object 类型的析构函数，由派生类型的析构函数自动调用，不能手动调用
Equals(object)	bool	是	否	把调用该方法的对象与另一个对象相比，如果它们相等，就返回 true。默认的实现代码会查看其对象参数是否引用了同一个对象(因为对象是引用类型)。如果想以不同方式来比较对象，则可以重写该方法，例如，比较两个对象的状态
Equals(object, object)	bool	否	是	这个方法比较传送给它的两个对象，看看它们是否相等。检查时使用了 Equals(object) 方法。注意，如果两个对象都是空引用，这个方法就返回 true
ReferenceEquals(object, object)	bool	否	是	这个方法比较传送给它的两个对象，看看它们是不是同一个实例的引用
ToString()	String	是	否	返回一个对应于对象实例的字符串。默认情况下，这是一个类类型的限定名称，但可以重写它，给类类型提供合适的实现方式
MemberwiseClone()	object	否	否	通过创建一个新对象实例并复制成员，以复制该对象。成员复制不会得到这些成员的新实例。新对象的任何引用类型成员都将引用与源类相同的对象，这个方法是受保护的，所以只能在类或派生的类中使用

(续表)

方 法	返回类型	虚拟	静态	说 明
GetType()	System.Type	否	否	以 System.Type 对象的形式返回对象的类型
GetHashCode()	int	是	否	在需要此参数的地方，用作对象的散列函数，它返回一个以压缩形式标识对象状态的值

这些方法是.NET Framework中对象类型必须支持的基本方法，但我们可能从不使用其中某些类型（或者只在特殊情况下使用，如 GetHashCode()）。

在利用多态性时，GetType()是一个有用的方法，允许根据对象的类型来执行不同的操作，而不是像通常那样，对所有对象都执行相同的操作。例如，如果函数接受一个object类型的参数（表示可以给该函数传递任何信息），就可以在遇到某些对象时执行额外任务。结合使用 GetType()和typeof（这是一个C#运算符，可以把类名转换为System.Type对象），就可以进行比较，如下所示：

```
if (myObj.GetType() == typeof(MyComplexClass))
{
    // myObj is an instance of the class MyComplexClass.
}
```

返回的System.Type对象可以完成更多工作，这里不讨论它们。重写ToString()方法也是非常有效的，在对象的内容可以用一个人们能理解的字符串表示时，尤其如此。后续章节将反复讨论这些System.Object方法，现在就讨论到这里为止，后面在需要时再详细讨论。

9.3 构造函数和析构函数

在C#中定义类时，常常不需要定义相关的构造函数和析构函数，因为在编写代码时，如果没有提供它们，编译器会自动添加它们。但是，如有必要，可以提供自己的构造函数和析构函数，以便初始化对象和清理对象。

使用下述语法可以把一个简单的构造函数添加到类中：

```
class MyClass
{
    public MyClass()

    {

        // Constructor code.

    }
}
```

```
}
```

这个构造函数与包含它的类同名，且没有参数（使其成为类的默认构造函数），这是一个公共函数，所以类的对象可以使用这个构造函数进行实例化（详见第8章）。

也可以使用私有的默认构造函数，即不能用这个构造函数来创建这个类的对象实例（它是不可创建的，详见第8章）：

```
class MyClass
{
    private MyClass()

    {

        // Constructor code.

    }
}
```

```
}
```

最后，通过提供参数，也可以采用相同的方式给类添加非默认的构造函数，例如：

```
class MyClass
```

```
{
```

```
    public MyClass()
```

```
{
```

```
    // Default constructor code.
```

```
}
```

```
    public MyClass(int myInt)
```

```
{  
  
    // Nondefault constructor code (uses myInt).  
  
}  
  
}
```

可提供的构造函数的数量不受限制（当然不能耗尽内存，也不能有相同的参数集，所以“几乎不受限”更合适）。

使用略微不同的语法来声明析构函数。在.NET中使用的析构函数（由System.Object类提供）称为Finalize()，但这不是我们用于声明析构函数的名称。使用下面的代码，而不是重写Finalize()：

```
class MyClass  
{  
    ~MyClass()
```

```
{  
  
    // Destructor body.  
  
}  
  
}
```

类的析构函数由带有~前缀的类名来声明（构造函数也使用类名声明）。当进行垃圾回收时，就执行析构函数中的代码，释放资源。调用这个析构函数后，还将隐式地调用基类的析构函数，包括System.Object根类中的Finalize()调用。该技术可以让.NET Framework确保调用Finalize()，因为重写Finalize()是指基类调用需要显式地执行，这具有潜在危险（第10章将详细讨论如何调用基类的方法）。

构造函数的执行序列

如果在类的构造函数中执行多个任务，把这些代码放在一个地方是非常方便的，这与第6章论述的把代码放在函数中有相同的优势。使用一个方法就可以把代码放在一个地方（详见第10章），但C#提供了一种

更好的方式。任何构造函数都可以配置为在执行自己的代码前调用其他构造函数。

在讨论构造函数前，先看一下在默认情况下，创建类的实例时会发生什么。除了前面说过的便于把初始化代码集中起来之外，还要了解这些代码。在开发过程中，由于调用构造函数时出现错误，对象常常并没有按照预期的那样执行。发生构造函数调用错误常常是因为类继承结构中的某个基类没有正确实例化，或者没有正确地给基类构造函数提供信息。如果理解在对象生命周期的这个阶段发生的事情，将更利于解决此类问题。

为了实例化派生的类，必须实例化它的基类。而要实例化这个基类，又必须实例化这个基类的基类，这样一直到实例化 `System.Object`（所有类的根）为止。结果是无论使用什么构造函数实例化一个类，总是首先调用 `System.Object.Object()`。

无论在派生类上使用什么构造函数（默认的构造函数或非默认的构造函数），除非明确指定，否则就使用基类的默认构造函数（稍后将介绍如何改变这个行为）。下面介绍一个简短示例，来演示执行顺序。考虑下面的对象层次结构：

```
public class MyBaseClass
{
    public MyBaseClass()
    {
    }
    public MyBaseClass(int i)
    {
```

```
    }  
}  
public class MyDerivedClass : MyBaseClass  
{  
    public MyDerivedClass()  
    {  
    }  
    public MyDerivedClass(int i)  
    {  
    }  
    public MyDerivedClass(int i, int j)  
    {  
    }  
}
```

如果以下面的方式实例化MyDerivedClass:

```
MyDerivedClass myObj = new MyDerivedClass();
```

则执行顺序如下:

- 执行System.Object.Object()构造函数。
- 执行MyBaseClass.MyBaseClass()构造函数。
- 执行MyDerivedClass.MyDerivedClass()构造函数。

另外, 如果使用下面的语句:

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

则执行顺序如下：

- 执行System.Object.Object()构造函数。
- 执行MyBaseClass.MyBaseClass()构造函数。
- 执行MyDerivedClass.MyDerivedClass (int i) 构造函数。

最后，如果使用下面的语句：

```
MyDerivedClass myObj = new MyDerivedClass(4, 8);
```

则执行顺序如下：

- 执行System.Object.Object()构造函数。
- 执行MyBaseClass.MyBaseClass()构造函数。
- 执行MyDerivedClass.MyDerivedClass (int i, int j) 构造函数。

大多数情况下，这个系统都能正常工作。但是，有时需要对发生的事件进行更多控制。例如，在上面的实例化示例中，可能想得到如下所示的执行顺序：

- 执行System.Object.Object()构造函数。
- 执行MyBaseClass.MyBaseClass (int i) 构造函数。
- 执行MyDerivedClass.MyDerivedClass (int i, int j) 构造函数。

使用这个顺序，可以把使用int i参数的代码放在MyBaseClass (int i) 中，即MyDerivedClass (int i, int j) 构造函数要做的工作较少，只需要处理int j参数（假定int i参数在两种情况下的含义相同，虽然事情并非总是如此，但实际上我们常常做这样的安排）。只要愿意，C#就可以指定这种操作。

为此，只需使用构造函数初始化器，它把代码放在方法定义的冒号后面。例如，可以在派生类的构造函数定义中指定所使用的基类构造函数，如下所示：

```
public class MyDerivedClass : MyBaseClass
{
    ...
    public MyDerivedClass(int i, int j) : base(i)

    {
    }
}
```

其中，**base**关键字指定.NET实例化过程使用基类中具有指定参数的构造函数。这里使用了一个**int**参数（其值通过参数*i*传送给MyDerivedClass构造函数），所以将使用MyBaseClass（int i）。这么做将不会调用MyBaseClass()，而是执行本例前面列出的事件序列——也就是我们希望执行的事件序列。

也可以使用这个关键字指定基类构造函数的字面值，例如，使用MyDerivedClass的默认构造函数来调用MyBaseClass的非默认构造函数：

```
public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass() : base(5)
```

```
{  
}  
...  
}
```

这段代码将执行下述序列：

- 执行System.Object.Object()构造函数。
- 执行MyBaseClass.MyBaseClass (int i) 构造函数。
- 执行MyDerivedClass.MyDerivedClass()构造函数。

除了base关键字外，这里还可将另一个关键字this用作构造函数初始化器。这个关键字指定在调用指定的构造函数前，.NET实例化过程对当前类使用非默认的构造函数。例如：

```
public class MyDerivedClass : MyBaseClass  
{  
    public MyDerivedClass() : this(5, 6)  
  
    {  
    }  
    ...  
    public MyDerivedClass(int i, int j) : base(i)
```

```
{  
}  
}
```

使用MyDerivedClass.MyDerivedClass()构造函数，将得到如下执行顺序：

- 执行System.Object.Object()构造函数。
- 执行MyBaseClass.MyBaseClass（int i）构造函数。
- 执行MyDerivedClass.MyDerivedClass（int i, int j）构造函数。
- 执行MyDerivedClass.MyDerivedClass()构造函数。

唯一的限制是使用构造函数初始化器只能指定一个构造函数。但如上一个示例所示，这并不是一个很严格的限制，因为我们仍可以构造相当复杂的执行顺序。

注意： 如果没有给构造函数指定构造函数初始化器，编译器就会自动添加base()。这会执行本节前面介绍的默认顺序。

注意在定义构造函数时，不要创建无限循环。例如：

```
public class MyBaseClass  
{  
    public MyBaseClass() : this(5)  
    {  
    }  
}
```

```
public MyBaseClass(int i) : this()  
{  
}  
}
```

使用上述任何一个构造函数，都需要首先执行另一个构造函数，而另一个构造函数要求首先执行原构造函数。这段代码可以编译，但如果尝试实例化MyBaseClass，就会得到一个SystemOverflow-Exception异常。

9.4 Visual Studio中的OOP工具

OOP在.NET Framework中是一个非常基础的主题，所以VS提供了几个工具来帮助开发OOP应用程序。本节就介绍其中的一些工具。

9.4.1 Class View窗口

第2章介绍了Solution Explorer窗口与Class View窗口共用相同的空间。这个窗口显示了应用程序中的类层次结构，可供查看我们使用的类的特性。对于上一节的示例项目，其Class View视图如图9-3所示。

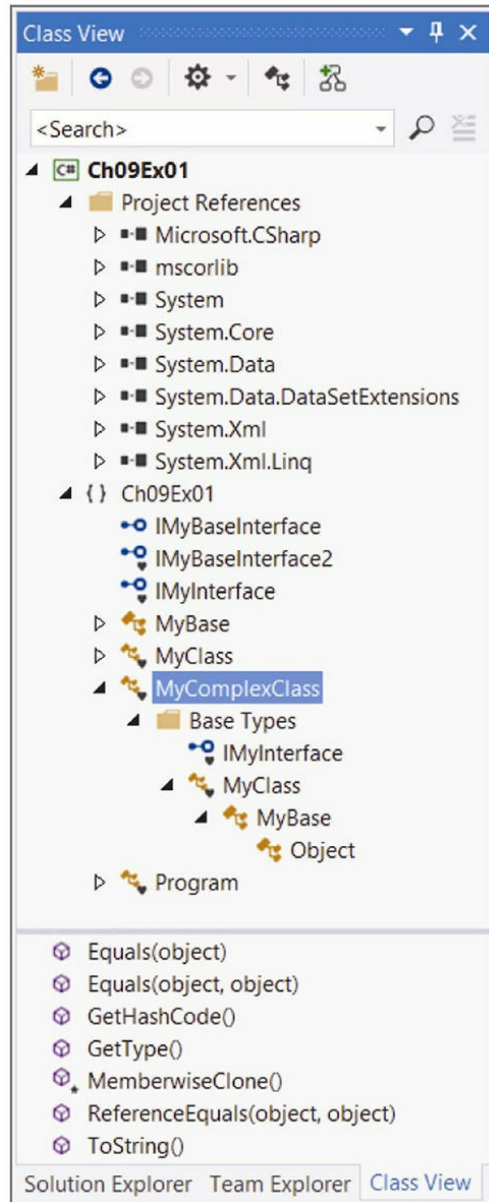


图9-3




这个窗口分为两部分，底下的一半显示了类型的成员。注意，图9-3显示的是选中Class View Settings下拉列表（位于Class View窗口的顶部）中的全部项以后的Class View窗口。

这里使用了许多符号，如表9-3所示。

表9-3 Class View窗口使用的图标

图 标	含 义	图 标	含 义	图 标	含 义
	项目		属性		事件
	名称空间		字段		委托
	类		结构		程序集

(续表)

图 标	含 义	图 标	含 义	图 标	含 义
	接口		枚举		
	方法		枚举项		

注意，其中一些图标用于类型定义而不是类定义，例如枚举和结构类型。

其中一些项的下面还有其他符号，表示它们的访问级别（公共项没有这样的符号），表9-4中列出了这些符号。

表9-4 Class View窗口中使用的其他图标

图 标	含 义	图 标	含 义	图 标	含 义
	私有的		受保护的		内部的

没有符号用于表示抽象、密封和虚拟项。

在这里除了可以查看信息外，还可以访问许多项的相关代码。双击某项，或者右击该项，然后选择Go To Definition，就可以查看项目中用于定义该项的代码（假定代码是可以查看的）。如果无法查看代码，例如不能访问基类型System.Object中的代码，就应该选择Browse

Definition，打开Object Browser视图（详见下一节）。

图9-3显示的另一项是Project References，它可以供查看项目引用了哪些程序集，本例的项目包含mscorlib和System中的核心.NET类型、System.Data中的数据访问类型和System.Xml中的XML操纵类型。这里的引用也是可以扩展的，显示这些程序集中包含的名称空间和类型。

Class View还可以查找代码中的类型和成员。其方法是，右击一项，选择Find All References，就会在Find Symbol Results窗口中打开搜索结果列表，该窗口位于屏幕底部，是Error List显示区域的一个选项卡。还可以使用Class View窗口对项进行重命名。在重命名时，可以重命名代码中出现的项的引用。也就是说，没有理由在类名中出现拼写错误，因为我们可以随时修改它们。

另外，使用Call Hierarchy视图可以在代码中导航。在Class View窗口中通过选择View Call Hierarchy，右击菜单项就可以访问Call Hierarchy窗口。这个功能非常适于查看类成员彼此之间的交互方式，参见下一章。

9.4.2 对象浏览器

对象浏览器（Object Browser）是Class View窗口的扩展版本，可以查看项目中能使用的其他类，甚至可以查看外部的类。可以自动（如上一节的情况）或手动（通过View|Object Browser）进入这个窗口。这个视图显示在主窗口中，可以采用与Class View窗口相同的方式浏览该视图。

这个窗口显示了与Class View窗口相同的信息，还显示了.NET类型的其他信息。选中某项，还可以在第三个窗口中获得该项的信息，如图9-4所示。

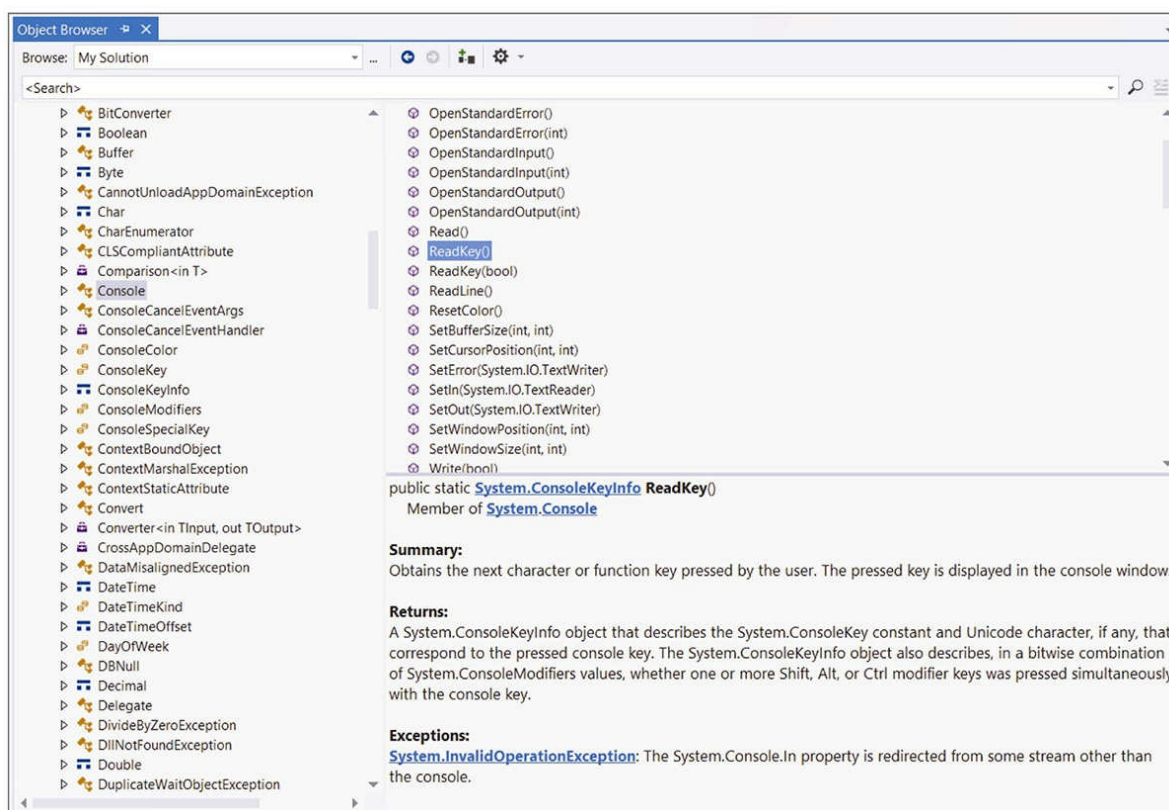


图9-4

在图9-4中，选中了Console类的ReadKey()方法（Console在mscorlib程序集的System名称空间中）。右下角的信息窗口显示了方法签名、该方法所属的类和方法功能的小结。在研究.NET类型时，或者了解某个类的用途时，这些信息非常有用。

还可以在自己创建的类型中使用这个信息窗口。对Ch09Ex01中的代码进行如下修改：

```
///<summary>
```

```
/// This class contains my program!
```

```
///</summary>
```

```
class Program
{
    static void Main(string[] args)
    {
        MyComplexClass myObj = new MyComplexClass();
        WriteLine(myObj.ToString());
        ReadKey();
    }
}
```

然后返回到对象浏览器，就会看到这些变化反映在信息窗口中。这是XML文档说明的一个示例，本书不讨论XML文档说明，但读者有闲暇时间时，应学习这个主题。

注意：如果手工修改上面的代码，只要键入3个斜杠///，IDE就会添加输入的其他内容。它会分析应用于XML文档说明的代码，建立基本的XML文档说明。显然，这进一步证明了VS是一个功能十分强大的工具。

9.4.3 添加类

VS包含可以加速执行某些常见任务的工具，其中一些可以应用于OOP。有一个Add New Item Wizard工具可以为项目快速添加新类，且需要键入的代码数量最少。

通过单击Project|Add New Item菜单项，或在Solution Explorer窗口中右击项目，选择相应的项，可以打开该工具。采用其中任意一种方式，都会打开一个对话框，在该对话框中，可以选择要添加的项。要添加一个类，可以在模板窗口中选择Class项，如图9-5所示，为包含类的文件提供一个文件名，再单击Add按钮。所创建的类就以所提供的文件名命名。

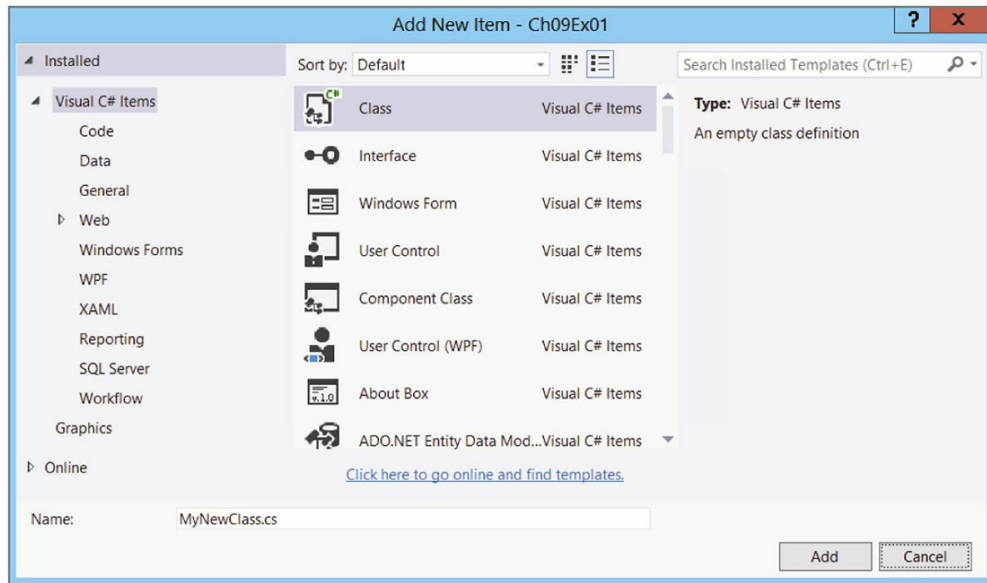


图9-5

在本章前面的示例中，我们在Program.cs文件中手动添加类定义。把类放在独立的文件中，常常可以更轻松地跟踪类。打开Ch09Ex01项目后，在Add New Item对话框中输入信息，就会在MyNewClass.cs中生成下列代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Ch09Ex01
{
    class MyNewClass
    {
    }
}
```

```
}
```

`MyNewClass`类在入口点类`Program`所在的名称空间中定义，所以可以在代码中使用它，就像是在相同的文件中定义一样。从代码中可以看出，生成的类不包含构造函数。如果类定义没有包含构造函数，编译器就会在编译代码时自动添加一个默认构造函数。

9.4.4 类图

还没有介绍的VS的一个强大功能是从代码中生成类图，并使用类图修改项目。VS中的类图编辑器可以很方便地为代码生成类似于UML的图。为描述这个功能，下面的示例将为前面创建的`Ch09Ex01`项目生成类图。

试一试：生成类图

- (1) 打开本章前面创建的`Ch09Ex01`项目。
- (2) 在Solution Explorer窗口中，右击`Ch09Ex01`项目，在上下文菜单中选择View|View Class Diagram菜单项。
- (3) 此时会显示一个类图`ClassDiagram1.cd`。
- (4) 单击`IMyInterface`“棒棒糖”，在Properties窗口中，把它的Position属性改为Right。

(5) 右击MyBase，从上下文菜单中选择Show Base Type选项。

(6) 拖动图中的对象，生成较美观的布局。完成这些步骤后，类图将如图9-6所示。

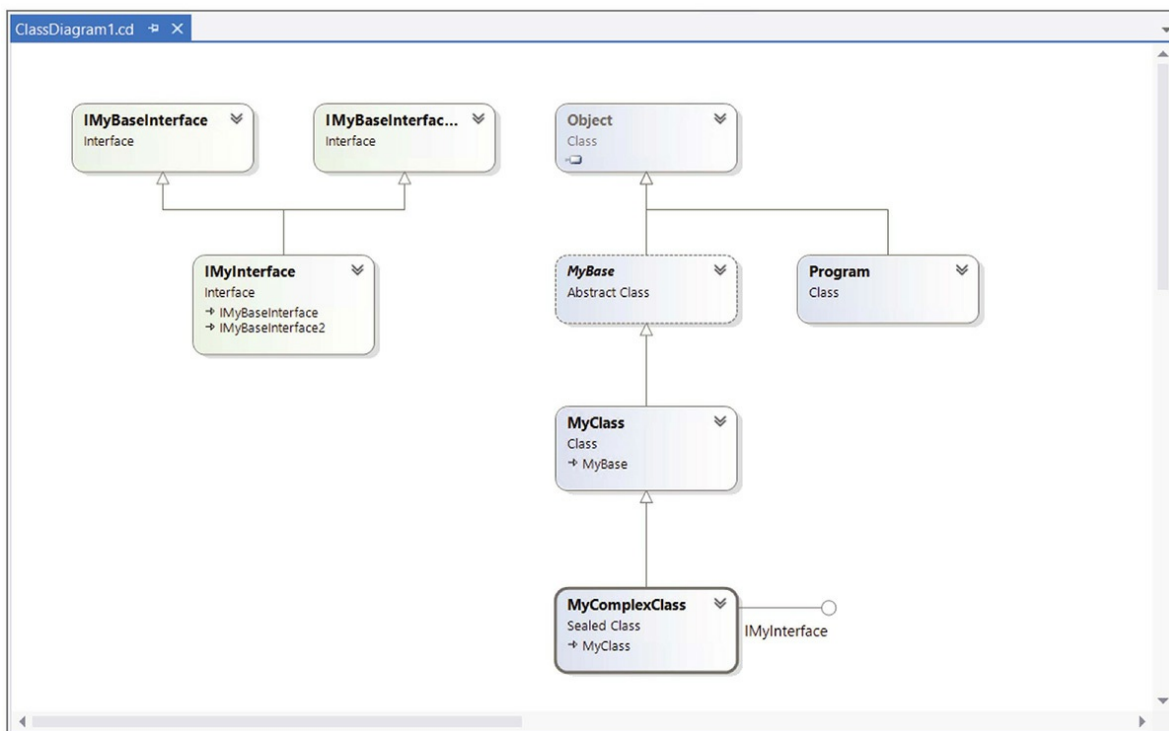


图9-6

示例说明

本例毫不费力地创建了一个与UML图（见图9-2）非常类似的类图，下面的特性得到了证明：

- 类显示为蓝色框，其中包含类的名称和类型。
- 接口显示为绿色框，其中包含接口的名称和类型。
- 继承用白色箭头表示，某些情况下，类框中包含文本。
- 实现接口的类有“棒棒糖”图标。

- 抽象类显示为虚点外框，名称显示为斜体。
- 密封类显示为粗黑外框。

单击一个对象会在屏幕底部的Class Details窗口中显示其他信息（如果Class Details窗口没有显示出来，可以右击一个对象，然后选择Class Details）。可以在此查看（和修改）类成员，还可以在Properties窗口中修改类的信息。

在Toolbox中，可以给图添加新项，例如类、接口和枚举等，定义图中对象之间的关系。此时，新项的代码会自动生成。

9.5 类库项目

除了在项目中把类放在不同的文件中之外，还可以把它们放在完全不同的项目中。如果一个项目只包含类（以及其他相关的类型定义，但没有入口点），该项目就称为类库。

类库项目编译为.dll程序集，在其他项目中添加对类库项目的引用，就可以访问它的内容，这可以是（也可以不是）同一个解决方案的一部分。这扩展了对象提供的封装性，因为修改和更新类库不会影响使用它们的其他项目。这意味着，可以方便地升级类提供的服务（会影响多个使用这些类的应用程序）。

下面看一个类库项目的示例和一个利用该类库项目包含的类的独立项目。

试一试：使用类库：**Ch09ClassLib**和**Ch09Ex02\Program.cs**

（1）在C:\BegVCSharp\Chapter09目录中创建一个Class Library类型的新项目Ch09ClassLib，如图9-7所示。

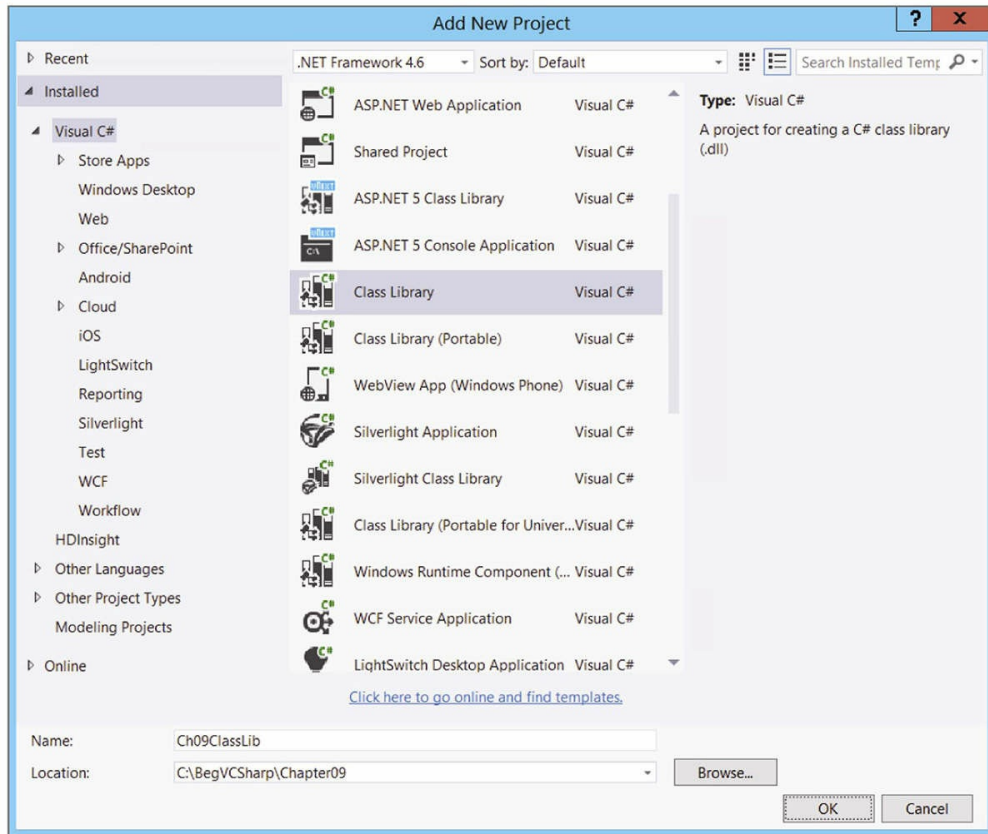


图9-7

(2) 把文件Class1.cs重命名为MyExternalClass.cs（在Solution Explorer窗口中右击该文件，然后选择Rename来重命名该文件名）。在弹出的对话框中单击Yes。

(3) MyExternalClass.cs中的代码随之自动改变，以反映类名的改变：

```
public class MyExternalClass
```

```
{  
}
```

(4) 使用文件名MyInternalClass.cs给项目添加一个新类。

(5) 修改代码，显式地指定类MyInternalClass是内部类：

internal

```
class MyInternalClass  
{  
}
```

(6) 编译项目（注意这个项目没有入口点，所以不能像通常那样运行它——可以选择Build|Build Solution菜单项来生成它）。

(7) 在C:\BegVCSharp\Chapter09目录中创建一个新的控制台应用程序项目Ch09Ex02。

(8) 选择Project|Add Reference菜单项，或者在Solution Explorer窗口中右击References，选择相同的选项。

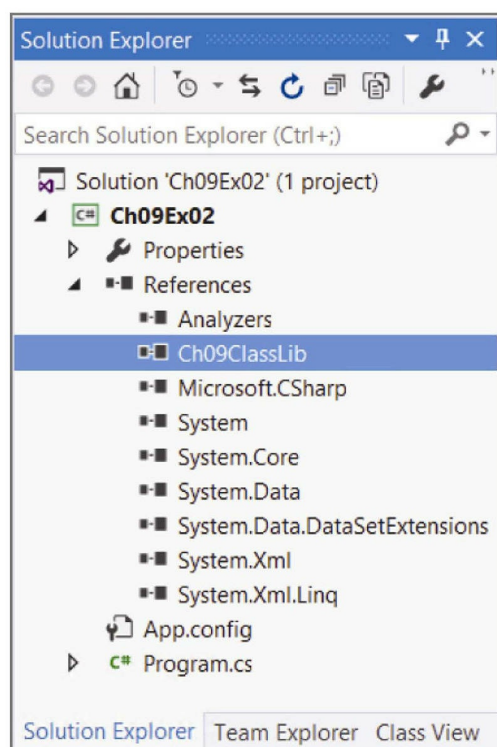


图9-8

(9) 单击Browse选项卡，导航到C:\BegVCSharp\Chapter09\Chapter09\Ch09ClassLib\bin\Debug\，双击Ch09ClassLib.dll。

(10) 完成上述操作后，检查是否已将引用添加到Solution Explorer窗口中，如图9-8所示。

(11) 打开Object Browser窗口，检查新引用，看看其中包含的对象，结果如图9-9所示。

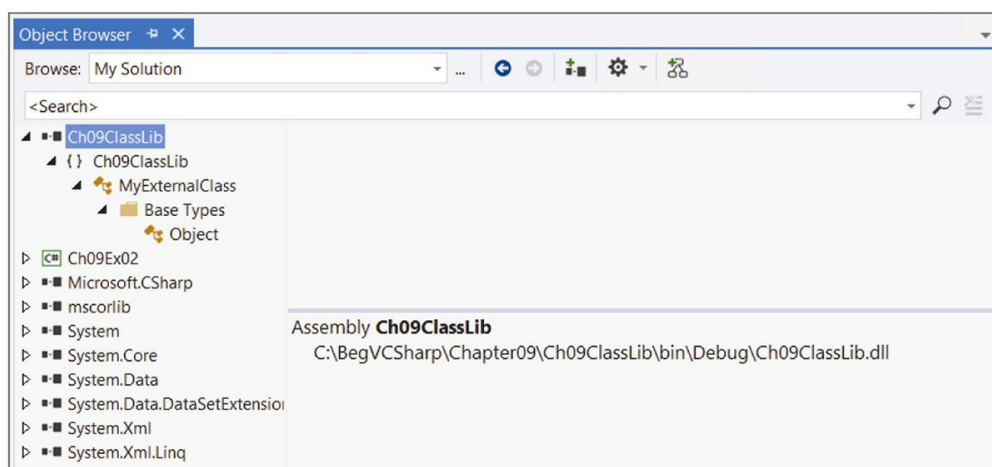


图9-9

(12) 修改Program.cs中的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;
```

```
using Ch09ClassLib;

namespace Ch09Ex02
{
    class Program
    {
        static void Main(string[] args)
        {
            MyExternalClass myObj = new MyExternalClass();

            WriteLine(myObj.ToString());

            ReadKey();
        }
    }
}
```

(13) 运行应用程序，结果如图9-10所示。

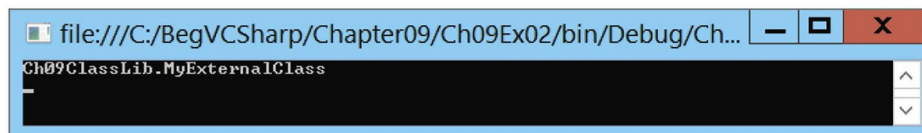


图9-10

示例说明

这个示例创建了两个项目，一个是类库项目，另一个是控制台应用程序项目。类库项目Ch09ClassLib包含两个类：**MyExternalClass**（可公开访问）和**MyInternalClass**（只能在内部访问）。注意，默认情况下，会隐式将类确定为供内部访问，因为它没有访问修饰符。最好明确指定可访问性，因为这会使代码更便于理解，所以指令增加了**internal**关键字。控制台应用程序项目Ch09Ex02包含利用类库项目的简单代码。

注意：当应用程序使用外部库中定义的类时，可以把该应用程序称为库的客户应用程序。使用所定义的类的代码一般简称为客户代码。

为使用Ch09ClassLib中的类，在控制台应用程序中添加了对Ch09ClassLib.dll的引用。对于这个示例，该引用指向类库的输出文件，不过也可以把这个文件复制到Ch09Ex02的本地位置，以便继续开发类库，而不影响控制台应用程序。为用新类库项目替换旧版本的程序集，只需用新生成的DLL文件覆盖旧文件即可。

添加引用后，就可以使用对象浏览器查看可用的类。因为类**MyInternalClass**是内部的，所以在对象浏览器窗口中看不到这个类——

它不能由外部项目访问。但是，MyExternalClass是可供访问的，这是我们在控制台应用程序中使用的类。

可以把控制台应用程序中的代码替换为使用内部类的代码，如下所示：

```
static void Main(string[] args)
{
    MyInternalClass myObj = new MyInternalClass();

    WriteLine(myObj.ToString());
    ReadKey();
}
```

如果试图编译这段代码，就会产生如下编译错误：

```
'Ch09ClassLib.MyInternalClass'
    is inaccessible due to its protection level
```

利用外部程序集中的类的技术是使用C#和.NET Framework编程的关键。实际上，使用.NET Framework中的任何类，也就是在利用外部程序集中的类，因为它们的处理方式是相同的。

9.6 接口和抽象类

本章介绍了如何创建接口和抽象类（现在不考虑其成员，第10章会讲述类的成员）。这两种类型在许多方面都十分类似，所以应看一下它们的相似和不同之处，看看哪些情况应使用什么技术。

首先讨论它们的相似之处。抽象类和接口都包含可以由派生类继承的成员。接口和抽象类都不能直接实例化，但可以声明这些类型的变量。如果这样做，就可以使用多态性把继承这两种类型的对象指定给它们的变量。接着通过这些变量来使用这些类型的成员，但不能直接访问派生对象的其他成员。

下面分析它们之间的区别。派生类只能继承自一个基类，即只能直接继承自一个抽象类（但可以用一个继承链包含多个抽象类）。相反，类可以使用任意多个接口。但这不会产生太大的区别——这两种情况取得的效果是类似的。只是采用接口的方式稍有不同。

抽象类可以拥有抽象成员（没有代码体，且必须在派生类中实现，否则派生类本身必须也是抽象的）和非抽象成员（它们拥有代码体，也可以是虚拟的，这样就可以在派生类中重写）。另一方面，接口成员必须都在使用接口的类上实现——它们没有代码体。另外，按照定义，接口成员是公共的（因为它们的目的是在外部使用），但抽象类的成员可以是私有的（只要它们不是抽象的）、受保护的、内部的或受保护的内部成员（其中受保护的内部成员只能在应用程序的代码或派生类中访问）。此外，接口不能包含字段、构造函数、析构函数、静态成员或常量。

注意：抽象类主要用作对象系列的基类，这些对象共享某些主要特性，例如共同的目的和结构。接口则主要用于类，这些类存在根本性的区别，但仍可以完成某些相同的任务。

例如，假定有一个对象系列表示火车，基类Train包含火车的核心定义，例如车轮的规格和引擎的类型（可以是蒸汽发动机、柴油发动机等）。但这个类是抽象的，因为并没有“一般的”火车。为创建一辆实际的火车，需要给该火车添加特性。为此，派生一些类，例如PassengerTrain、FreightTrain和424DoubleBogey等，如图9-11所示。

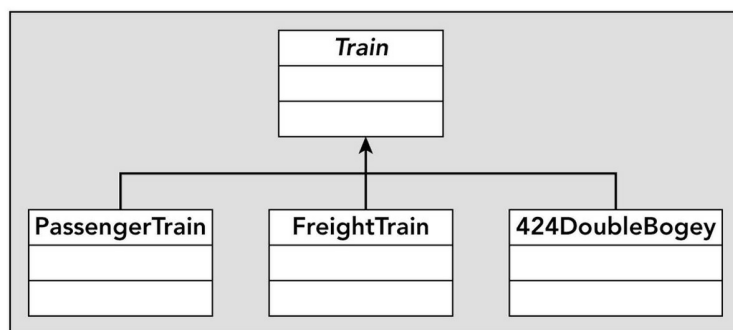


图9-11

也可以用相同的方式来定义汽车对象系列，使用Car抽象基类，其派生类有Compact、SUV和PickUp。Car和Train甚至可以派生于一个相同的基类Vehicle，如图9-12所示。

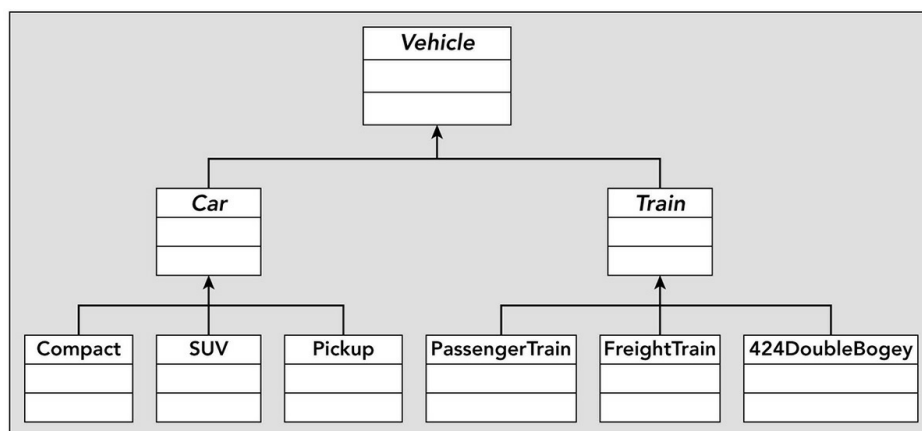


图9-12

现在，层次结构中的一些类共享相同的特性，这是因为它们的目的是一样的，而不只是因为它们派生于同一个基类。例如，**PassengerTrain**、**Compact**、**SUV**和**Pickup**都可以运送乘客，所以它们都拥有**IPassengerCarrier**接口。**FreightTrain**和**Pickup**可以运送重载货物，所以它们都拥有**IHeavyLoadCarrier**接口，如图9-13所示。

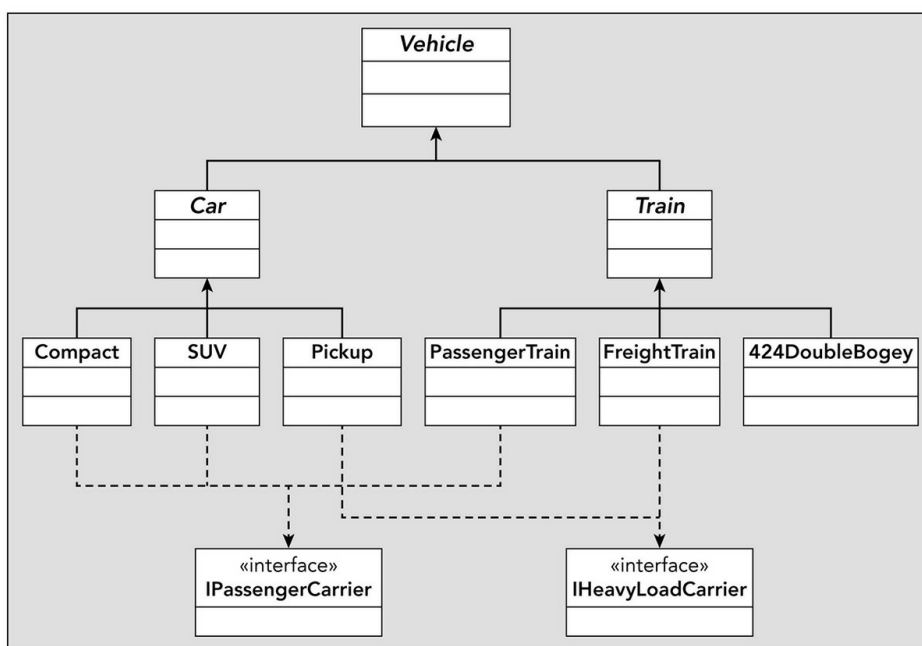


图9-13

在进行更详细的分解之前，把对象系统以这种方式进行分解，可以清晰地看到哪种情形适合使用抽象类，哪种情形适合使用接口。只使用接口或只使用抽象继承，就得不到这个示例的结果。

9.7 结构类型

第8章提到过，结构和类非常相似，但结构是值类型，而类是引用类型。这意味着什么？最简明的方式是用一个示例来说明。

试一试：类和结构：**Ch09Ex03\Program.cs**

(1) 在C:\BegVCSharp\Chapter09目录中创建一个新的控制台应用程序项目Ch09Ex03。

(2) 修改代码，如下所示：

```
namespace Ch09Ex03
{
    class MyClass

    {

        public int val;
```

```
}
```

```
struct myStruct
```

```
{
```

```
    public int val;
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        MyClass objectA = new MyClass();
```

```
MyClass objectB = objectA;
```

```
objectA.val = 10;
```

```
objectB.val = 20;
```

```
myStruct structA = new myStruct();
```

```
myStruct structB = structA;
```

```
structA.val = 30;
```



```
structB.val = 40;
```

```
WriteLine("objectA.val = {objectA.val}");
```

```
WriteLine("objectB.val = {objectB.val}");
```

```
WriteLine("structA.val = {structA.val}");
```

```
WriteLine("structB.val = {structB.val}");
```

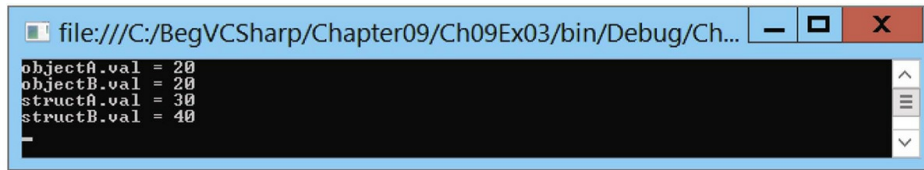
```
ReadKey();
```

```
}
```

```
}
```

```
}
```

(3) 运行应用程序，结果如图9-14所示。



```
file:///C:/BegVCSharp/Chapter09/Ch09Ex03/bin/Debug/Ch...
objectA.val = 20
objectB.val = 20
structA.val = 30
structB.val = 40
```

图9-14

示例说明

这个应用程序包含两个类型定义：一个是结构myStruct的定义，它有一个公共int字段val；另一个是类MyClass的定义，它包含一个相同的字段（第10章介绍类的成员，如字段，现在只要知道它们的语法是相同的即可）。接着对这两种类型的实例执行相同的操作：

- (1) 声明类型的一个变量。
- (2) 在这个变量中创建该类型的新实例。
- (3) 声明类型的第二个变量。
- (4) 将第一个变量赋给第二个变量。
- (5) 在第一个变量的实例中，给val字段赋予一个值。
- (6) 在第二个变量的实例中，给val字段赋予一个值。
- (7) 显示这两个变量的val字段值。

尽管对这两种类型的变量执行了相同的操作，但结果是不同的。在

显示val字段的值时，两个object类型具有相同的值，而结构类型有不同的值。为什么会这样？

对象是引用类型。把对象赋给变量时，实际上是把带有一个指针的变量赋给了该指针所指向的对象。在实际代码中，指针是内存中的一个地址。这种情况下，地址是内存中该对象所在的位置。用下面的代码行把第一个对象引用赋给类型为MyClass的第二个变量时，实际上是复制了这个地址。

```
MyClass objectB = objectA;
```

这样两个变量就包含同一个对象的指针。

结构是值类型。其变量并不是包含结构的指针，而是包含结构本身。在用下面的代码把第一个结构赋给类型为myStruct的第二个变量时，实际上是把第一个结构的所有信息复制到第二个结构中。

```
myStruct structB = structA;
```

这个过程与本书前面介绍的简单变量类型（如int）是一样的。最终结果是两个结构类型变量包含不同的结构。使用指针的全部技术隐藏在托管C#代码中，这使得代码更简单。使用C#中的不安全代码可以执行低级操作，如指针操作，但这是一个比较高级的主题，这里不予讨论。

9.8 浅度和深度复制

从一个变量到另一个变量按值复制对象，而不是按引用复制对象（即以与结构相同的方式复制）可能非常复杂。因为一个对象可能包含许多其他对象的引用，例如字段成员等，这将涉及许多繁杂的处理。把每个成员从一个对象复制到另一个对象中可能不会成功，因为其中一些成员可能是引用类型。

.NET Framework考虑了这个问题。简单地按照成员复制对象可以通过派生于System.Object的MemberwiseClone()方法来完成，这是一个受保护的方法，但很容易在对象上定义一个调用该方法的公共方法。这个方法提供的复制功能称为浅度复制（shallow copy），因为它并未考虑引用类型成员。因此，新对象中的引用成员就会指向源对象中相同成员引用的对象，在许多情况下这并不理想。如果要创建成员的新实例（复制值，而不复制引用），此时需要使用深度复制（deep copy）。

可以实现一个ICloneable接口，以标准方式进行深度复制。如果使用这个接口，就必须实现它包含的Clone()方法。这个方法返回一个类型为System.Object的值。我们可以采用各种处理方式，实现所选的任何一个方法体来得到这个对象。如果愿意，就可以进行深度复制（但不是必须执行深度复制，所以如果执行浅度复制更合适，就可以执行浅度复制）。对于该方法应该返回什么，并不存在规则或限制，所以很多人建议不要使用它。这些人建议实现自己的深度复制方法。第11章将详细介绍这个接口。

9.9 练习

(1) 下面的代码存在什么错误？

```
public sealed class MyClass
{
    // Class members.
}
public class myDerivedClass : MyClass
{
    // Class members.
}
```

(2) 如何定义不能创建的类（noncreatable class）？

(3) 为什么不能创建的类仍旧有用？如何利用它们的功能？

(4) 在类库项目**Vehicles**中编写代码，实现本章前面讨论的**Vehicle**对象系列，其中有9个对象和两个接口需要实现。

(5) 创建一个控制台应用程序项目**Traffic**，它引用**Vehicles.dll**（在练习题（4）中创建），其中包括函数**AddPassenger()**，它接受任何带有**IPassengerCarrier**接口的对象。要证明代码可以运行，使用支持这个接口的每个对象实例调用该函数，在每个对象上调用派生于**System.Object**的**ToString()**方法，并将结果输出到屏幕上。

附录A给出了练习答案。

9.10 本章要点

主题	要点
类和接口定义	类用class关键字定义，接口用interface关键字定义。可以使用public和internal关键字来定义类和接口的可访问性，类可以定义为abstract或sealed，以便控制继承性。父类和父接口在一个用逗号分隔的列表中指定，放在类或接口名和一个冒号的后面。在类定义中，只能指定一个父类，且必须是列表中的第一项
构造函数和析构函数	类自动带有默认的构造函数和析构函数的实现代码，我们很少需要提供自己的析构函数。可以使用可访问性、类名和可能需要的任何参数来定义构造函数。基类的构造函数在派生类的构造函数之前执行，使用this和base构造函数初始化器关键字，可以控制类中这些构造函数的执行顺序
类库	可以创建只包含类定义类库项目。这些项目不能直接执行，而必须通过客户代码在可执行程序中访问。VS为创建、修改和测试类提供了各种工具
类系列	类可以组合为系列，以提供公共的操作或共享公共特性。为此，可从共享的基类（可以是抽象的）中继承，或者实现接口
结构定义	结构的定义方式与类非常类似，但结构是值类型，而类是引用类型
复制对象	复制对象时，必须注意应复制该对象包含的其他对象，而不是仅复制这些对象的引用。复制引用称为浅度复制，而完全复制称为深度复制。可用ICloneable接口作为一个框架，提供类定义中的深度复制功能

第10章 定义类成员

本章内容：

- 如何定义类成员
- 如何控制类成员的继承
- 如何定义嵌套的类
- 如何实现接口
- 如何使用部分类定义
- 如何使用Call Hierarchy窗口

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 10 Code后，可以找到与本章示例对应的单独文件。

本章继续讨论在C#中如何定义类，主要介绍的是如何定义字段、属性和方法等类成员。首先介绍每种类型需要的代码，以及如何生成相应代码的结构。还将论述如何通过编辑成员的属性，来快速修改这些成员。

介绍完成员定义的基础知识后，将讨论一些比较高级的成员技术：

隐藏基类成员，调用重写的基类成员、嵌套的类型定义和部分类定义。

最后将理论付诸实践，创建一个类库，以便在后续章节中使用它。

10.1 成员定义

在类定义中，也提供了该类中所有成员的定义，包括字段、方法和属性。所有成员都有自己的访问级别，用下面的关键字之一来定义：

- **public**——成员可以由任何代码访问。
- **private**——成员只能由类中的代码访问（如果没有使用任何关键字，就默认使用这个关键字）。
- **internal**——成员只能由定义它的程序集（项目）内部的代码访问。
- **protected**——成员只能由类或派生类中的代码访问。

后两个关键字可以结合使用，所以也有**protected internal**成员。它们只能由项目（更确切地讲，是程序集）中派生类的代码来访问。

也可以使用关键字**static**来声明字段、方法和属性，这表示它们是类的静态成员，而不是对象实例的成员，详见第8章。

10.1.1 定义字段

用标准的变量声明格式（可以进行初始化）和前面介绍的修饰符来定义字段，例如：

```
class MyClass
{
    public int MyInt;
```

```
}
```

注意：.NET Framework中的公共字段以PascalCasing形式来命名，而不是camelCasing。这里使用的就是这种大小写形式，所以上面的字段称为MyInt而不是myInt。这仅是推荐使用的一种名称大小写形式，但它的意义非常重大。私有字段没有推荐的名称大小写模式，它们通常使用camelCasing来命名。

字段也可以使用关键字readonly，表示这个字段只能在执行构造函数的过程中赋值，或由初始化赋值语句赋值。例如：

```
class MyClass
{
    public readonly

int MyInt = 17;
}
```

如本章的导言所述，可使用static关键字将字段声明为静态，例如：

```
class MyClass
{
```

```
    public static int MyInt;  
}
```

静态字段必须通过定义它们的类来访问（在上面的示例中，是 `MyClass.MyInt`），而不是通过这个类的对象实例来访问。另外，可使用关键字 `const` 来创建一个常量值。按照定义，`const` 成员也是静态的，所以不需要使用 `static` 修饰符（实际上，使用 `static` 修饰符会产生一个错误）。

10.1.2 定义方法

方法使用标准函数格式、可访问性和可选的 `static` 修饰符来声明。例如：

```
class MyClass  
{  
    public string GetString() => return "Here is a string  
  
    .";  
}
```

注意： 与公共字段一样，.NET Framework 中的公共方法也采用 PascalCasing 形式来命名。

注意，如果使用了static关键字，这个方法就只能通过类来访问，不能通过对象实例来访问。也可以在方法定义中使用下述关键字：

- **virtual**——方法可以重写。
- **abstract**——方法必须在非抽象的派生类中重写（只用于抽象类中）。
- **override**——方法重写了一个基类方法（如果方法被重写，就必须使用该关键字）。
- **extern**——方法定义放在其他地方。

下面的代码是方法重写的一个示例：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        // Base implementation.
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override

void DoSomething()
{
    // Derived class implementation, overrides base implementa
}
```

```
}
```

如果使用了`override`，也可以使用`sealed`来指定在派生类中不能对这个方法做进一步修改，即这个方法不能由派生类重写。例如：

```
public class MyDerivedClass : MyBaseClass
{
    public override sealed

void DoSomething()
{
    // Derived class implementation, overrides base implementa
}
}
```

使用`extern`可以在项目外部提供方法的实现代码。这是一个高级论题，这里不做详细讨论。

10.1.3 定义属性

属性定义方式与字段类似，但包含的内容比较多。如前所述，属性比字段复杂，因为它们在修改状态前还可以执行一些额外操作，实际上，它们可能并不修改状态。属性拥有两个类似于函数的块，一个块用于获取属性的值，另一个块用于设置属性的值。

这两个块也称为访问器，分别用`get`和`set`关键字来定义，可以用于

控制属性的访问级别。可以忽略其中的一个块来创建只读或只写属性（忽略`get`块创建只写属性，忽略`set`块创建只读属性）。当然，这仅适用于外部代码，因为类中的其他代码可以访问这些代码块能访问的数据。还可以在访问器上包含可访问修饰符，例如使`get`块变成公共的，使`set`块变成受保护的。至少包含其中一个块，属性才是有效的（既不能读取也不能修改的属性没有任何用处）。

属性的基本结构包括标准的可访问修饰符（`public`、`private`等），后跟类名、属性名和`get`块（或`set`块，或者`get`块和`set`块，其中包含属性处理代码），例如：

```
public int MyIntProp
{

    get

}

// Property get code.
```

```
}
```

```
set
```

```
{
```

```
// Property set code.
```

```
}
```

```
}
```


类外部的代码不能直接访问这个myInt字段，因为其访问级别是私有的。外部代码必须使用属性来访问该字段。set函数采用类似方式把一个值赋给字段。这里可使用关键字value表示用户提供的属性值：

```
// Field used by property.
private int myInt;
// Property.
public int MyIntProp
{
    get { return myInt; }
    set { myInt = value; }
}
```

value等于类型与属性相同的一个值，所以如果属性和字段使用相同的类型，就不必考虑数据类型转换了。

这个简单属性只是用来阻止对myInt字段的直接访问。在对操作进行更多控制时，属性的真正作用才能发挥出来。例如，使用下面的代码实现set块：

```
set
{
    if (value >= 0 && value<= 10)
```

```
        myInt = value;

    }
}
```

只有赋给属性的值在0~10之间，才会修改myInt。此时，要做一个重要的设计选择：如果使用了无效值，该怎么办？有4种选择：

- 什么也不做（如上述代码所示）。
- 给字段赋默认值。
- 继续执行，就像没发生错误一样，但记录下该事件，以备将来分析。
- 抛出异常。

一般情况下，后两个选择效果较好，选择哪个选项取决于如何使用类，以及给类的用户授予多少控制权。抛出异常给用户提供的控制权相当大，可以让他们了解发生了什么情况，并做适当的响应。为此可使用System名称空间中的标准异常，例如：

```
set
{
    if (value >= 0 && value<= 10)
        myInt = value;
    else
```

```
throw (new ArgumentOutOfRangeException("MyIntProp", value
```

```
"MyIntProp must be assigned a value between 0 and 10."
```

```
}
```

这可以在使用属性的代码中通过try...catch...finally逻辑加以处理，详见第7章。

记录数据（例如记录到文本文件中）比较有效，例如可以用在不应出错的产品代码中。它们允许开发人员检查性能，如有必要，还可以调试现有的代码。

属性可以使用virtual、override和abstract关键字，就像方法一样，但这几个关键字不能用于字段。最后，如上所述，访问器可以有自己的可访问性，例如：

```
// Field used by property.  
private int myInt;  
// Property.  
public int MyIntProp  
{  
    get { return myInt; }  
}
```

protected set

```
{ myInt = value; }  
}
```

只有类或派生类中的代码才能使用set访问器。

访问器可以使用的访问修饰符取决于属性的可访问性，访问器的可访问性不能高于它所属的属性，也就是说，私有属性对它的访问器不能包含任何可访问修饰符，而公共属性可以对其访问器使用所有的可访问修饰符。

C# 6引入了一个名为“基于表达式的属性”的功能。类似于第6章讨论的基于表达式的方法，这个功能可以把属性的定义减少为一行代码。例如，下面的属性对一个值进行数学计算，可以使用Lambda箭头后跟等式来定义：

```
// Field used by property.  
private int myDoubledInt = 5;  
// Property.  
public int MyDoubledIntProp => (myDoubledInt * 2);
```

下面的示例将定义和使用字段、方法和属性。

试一试：使用字段、方法和属性：**Ch10Ex01**

(1) 在C:\BegVCSharp\Chapter10目录中创建一个新控制台应用程序Ch10Ex01。

(2) 使用Add Class快捷方式添加一个新类MyClass，这将在新文件MyClass.cs中定义这个新类。

(3) 修改MyClass.cs中的代码，如下所示：

```
public class MyClass
```

```
{
```

```
    public readonly string Name;
```

```
    private int intVal;
```

```
    public int Val
```

```
{
```

```
    get { return intVal; }
```

```
    set {
```

```
        if (value >= 0 && value<= 10)
```

```
            intVal = value;
```

```
        else
```

```
            throw (new ArgumentOutOfRangeException("Val", value,
```

```
"Val must be assigned a value between 0 and 10.")).
```

```
}
```

```
}
```

```
public override string ToString() => "Name: " + Name + "\n
```

```
}
```

```
private MyClass() : this("Default Name") { }
```

```
public MyClass(string newName)
```

```
{
```

```
    Name = newName;
```

```
    intVal = 0;
```

```
}
```

```
private int myDoubledInt;
```

```
public int myDoubledIntProp => (myDoubledInt * 2);
```

```
}
```


(4) 修改Program.cs中的代码，如下所示：

```
using static System.Console;
static void Main(string[] args)
{
    WriteLine("Creating object myObj...");

    MyClass myObj = new MyClass("My Object");

    WriteLine("myObj created.");

    for (int i = -1; i <= 0; i++)

    {

        try
```

```
{
```

```
WriteLine($"Attempting to assign {i} to myObj.Val...")
```

```
myObj.Val = i;
```

```
WriteLine($"Value {myObj.Val} assigned to myObj.Val.");
```

```
}
```

```
catch (Exception e)
```

```
{
```

```
    WriteLine($"Exception {e.GetType().FullName} thrown.");
```

```
    WriteLine($"Message:\n\"{e.Message}\"");
```

```
}
```

```
}
```

```
WriteLine("\nOutputting myObj.ToString()...");
```

```
WriteLine(myObj.ToString());
```

```
WriteLine("myObj.ToString() Output.");
```

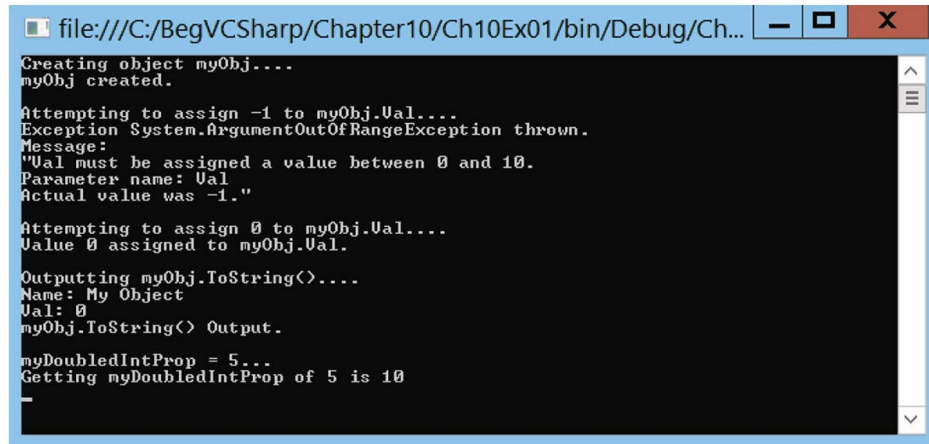
```
WriteLine("\nmyDoubledIntProp = 5...");
```

```
WriteLine($"Getting myDoubledIntProp of 5 is {myObj.myDoub
```

```
ReadKey();
```

```
}
```

(5) 运行应用程序，其结果如图10-1所示。



```
file:///C:/BegVCSharp/Chapter10/Ch10Ex01/bin/Debug/Ch...
Creating object myObj....
myObj created.
Attempting to assign -1 to myObj.Val....
Exception System.ArgumentOutOfRangeException thrown.
Message:
"Val must be assigned a value between 0 and 10.
Parameter name: Val
Actual value was -1."
Attempting to assign 0 to myObj.Val....
Value 0 assigned to myObj.Val.
Outputting myObj.ToString()....
Name: My Object
Val: 0
myObj.ToString() Output.
myDoubledIntProp = 5...
Getting myDoubledIntProp of 5 is 10
-
```

图10-1

示例的说明

Main()中的代码创建并使用在MyClass.cs中定义的MyClass类的实例。实例化这个类必须使用非默认的构造函数来进行，因为MyClass类的默认构造函数是私有的：

```
private MyClass() : this("Default Name") {}
```

注意，这里用this（"Default Name"）来保证，如果调用了该构造函数，Name就获取一个值。如果这个类用于派生一个新类，这就是可能的。必须这么做，因为不给Name字段赋值，就会在后面产生错误。

所使用的非默认构造函数把值赋给只读字段Name（只能在字段声明或在构造函数中给它赋值）和私有字段intVal。

接着，Main()试着给myObj（MyClass的实例）的Val属性两次赋值。使用for循环在两次迭代中赋值-1和0，使用try...catch结构检查抛出的任何异常。把-1赋给属性时，会抛出System.ArgumentOutOfRangeException类型的异常，catch块中的代码会把该异常的信息输出到

控制台窗口中。在下一个循环中，值0成功地赋给Val属性，通过这个属性再把值赋给私有字段intVal。

最后，使用重写的ToString()方法输出一个格式化的字符串，来表示对象的内容：

```
public override string ToString() => "Name: " + Name + "\r\n";
```

必须使用override关键字来声明这个方法，因为它重写了基类System.Object的虚拟方法ToString()。此处的代码直接使用属性Val，而不是私有字段intVal。没理由不以这种方式使用类中的属性，但这可能会对性能产生轻微影响（对性能的影响非常小，我们不会察觉到）。当然，使用属性也可以在属性中进行固有的有效性验证，这对类中的代码也是有好处的。

最后在MyClass.cs中创建只读属性myDoubledInt并设置为5。使用基于表达式的属性功能，返回乘以2后的值：

```
public int MyDoubledIntProp => (myDoubledInt * 2);
```

当使用myObj.myDoubledIntProp访问属性时，输出是2乘以5等于10，与预期相符。

10.1.4 重构成员

在添加属性时有一项很方便的技术，可以从字段中生成属性。下面是一个重构（refactoring）的示例，“重构”表示使用工具修改代码，而

不是手工修改。为此，只需要右击类图中的某个成员，或在代码视图中右击某个成员即可。

例如，如果MyClass类包含如下字段：

```
public string myString;
```

右击该字段，选择Quick Actions...，就会打开如图10-2所示的对话框。

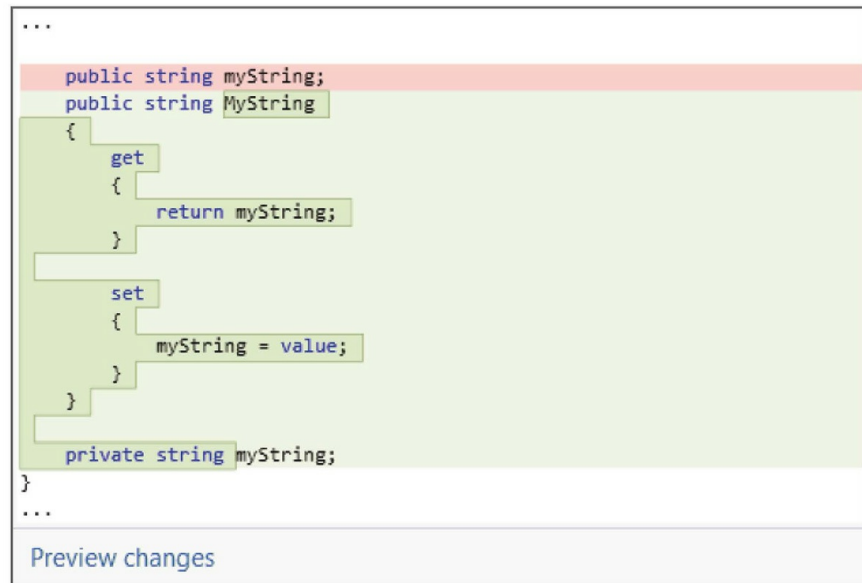


图10-2

接受默认选项，就会修改MyClass的代码，如下所示：

```
public string myString;
public string MyString
{
    get { return myString; }
    set { myString = value; }
```

```
}  
private string myString;
```

myString字段的可访问性变成private，同时创建了一个公共属性MyString，它自动链接到myString上。显然，这会减少为字段创建属性的时间。

10.1.5 自动属性

属性是访问对象状态的首选方式，因为它们禁止外部代码访问对象内部的数据存储机制的实现。属性还对内部数据的访问方式施加了更多控制，本章代码在多处体现了这一点。但是，一般以非常标准的方式定义属性，即通过一个公共属性来直接访问一个私有成员。其代码非常类似于上一节的代码，这是VS重构工具自动生成的。

重构功能肯定加快了键入速度，不过除此以外，C#另外提供了一种方式：自动属性。对于自动属性，可以用简化的语法声明属性，C#编译器会自动添加未键入的内容。确切地讲，编译器会声明一个用于存储属性的私有字段，并在属性的get和set块中使用该字段，我们不必考虑细节。

使用下面的代码结构就可以定义一个自动属性：

```
public int MyIntProp  
{  
    get;  
    set;
```



```
}
```

甚至可以在一行代码上定义自动属性，以便节省空间，而不会过度地降低属性的可读性：

```
public int MyIntProp { get; set; }
```

我们按照通常的方式定义属性的可访问性、类型和名称，但没有给`get`和`set`块提供实现代码。这些块的实现代码（和底层的字段）都由编译器提供。

注意： 使用Visual Studio中的支持代码片段，可以创建一个自动实现的属性模板。输入`prop`后按`Tab`键两次，就会自动创建`public int MyProperty {get; set;}。`

使用自动属性时，只能通过属性访问数据，不能通过底层的私有字段来访问，因为我们不知道底层私有字段的名称（该名称是在编译期间定义的）。但这并不是一个真正意义上的限制，因为可以直接使用属性名。自动属性的唯一限制是它们必须包含`get`和`set`存取器，无法使用这种方式定义只读或只写属性。但可以改变这些存取器的可访问性。例如，可采用如下方式创建一个外部只读属性：

```
public int MyIntProp { get; private set; }
```

此时，只能在类定义的代码中访问`MyIntProp`的值。

C# 6引入了两个与自动属性相关的新概念：只有`get`存取器的自动属

性，和自动属性的初始化器。在C# 6之前，自动属性需要set存取器，来限制不变数据类型的使用。不变数据类型的简单定义是，一旦创建，就不会改变状态。最著名的不变类型是System.String。使用不变的数据类型有很多优点，比如简化了并发编程和线程的同步。

并发编程和线程的同步是高级主题，本书不进一步讨论。然而一定要知道只有get存取器的自动属性。它们使用以下语法创建，注意不再需要set存取器：

```
public int MyIntProp { get; }
```

自动属性的初始化功能由以下声明字段的方式实现：

```
public int MyIntProp { get; } = 9;
```

10.2 类成员的其他主题

下面该讨论一些较高级的成员主题了。本节主要研究：

- 隐藏基类方法
- 调用重写或隐藏的基类方法
- 嵌套的类型定义

10.2.1 隐藏基类方法

当从基类继承一个（非抽象的）成员时，也就继承了其实现代码。如果继承的成员是虚拟的，就可以用`override`关键字重写这段实现代码。无论继承的成员是否为虚拟，都可以隐藏这些实现代码。这是很有用的，例如，当继承的公共成员不像预期的那样工作时，就可以隐藏它。

使用下面的代码就可以隐藏：

```
public class MyBaseClass
{
    public void DoSomething()
    {
        // Base implementation.
    }
}
```

```
public class MyDerivedClass : MyBaseClass
{
    public void DoSomething()

    {

        // Derived class implementation, hides base implementatio

    }

}
```

尽管这段代码可以正常运行，但它会生成一个警告，说明隐藏了一个基类成员。如果是无意间隐藏了一个需要使用的成员，此时就可以改正错误。如果确实要隐藏该成员，就可以使用new关键字显式地表明意图：

```
public class MyDerivedClass : MyBaseClass
{
```

new

```
public void DoSomething()  
{  
    // Derived class implementation, hides base implementation  
}  
}
```

其工作方式是完全相同的，但不会显示警告。此时应注意隐藏基类成员和重写它们的区别。考虑下面的代码：

```
public class MyBaseClass  
{  
    public virtual void DoSomething() => WriteLine("Base imp")  
  
}  
  
public class MyDerivedClass : MyBaseClass  
{  
    public override void DoSomething() => WriteLine("Derived i  
  
}
```

其中重写方法将替换基类中的实现代码，这样，下面的代码就将使

用新版本，即使这是通过基类类型进行的，情况也同样如此（使用多态性）：

```
MyDerivedClass myObj = new MyDerivedClass();  
MyBaseClass myBaseObj;  
myBaseObj = myObj;  
myBaseObj.DoSomething();
```

结果如下：

```
Derived imp
```

另外，还可以使用下面的代码隐藏基类方法：

```
public class MyBaseClass  
{  
    public virtual void DoSomething() => WriteLine("Base imp");  
}  
public class MyDerivedClass : MyBaseClass  
{  
    new  
  
    public void DoSomething() => WriteLine("Derived imp");  
}
```

基类方法不必是虚拟的，但结果是一样的，只需要修改上面代码中的一行即可。对于基类的虚拟方法和非虚拟方法而言，其结果如下：

Base imp

尽管隐藏了基类的实现代码，但仍可以通过基类访问它。

10.2.2 调用重写或隐藏的基类方法

无论是重写成员还是隐藏成员，都可以在派生类的内部访问基类成员。这在许多情况下都是很有用的，例如：

- 要对派生类的用户隐藏继承的公共成员，但仍能在类中访问其功能。
- 要给继承的虚拟成员添加实现代码，而不是简单地用重写的新实现代码替换它。

为此，可使用**base**关键字，它表示包含在派生类中的基类的实现代码（在控制构造函数时，其用法是类似的，如第9章所述），例如：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        // Base implementation.
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
```

```
{  
    // Derived class implementation, extends base class implem  
    base.DoSomething();  
  
    // More derived class implementation.  
}  
}
```

这段代码在MyDerivedClass包含的DoSomething()方法中，执行包含在MyBaseClass中的DoSomething()版本，MyBaseClass是MyDerivedClass的基类。因为base使用的是对象实例，所以在静态成员中使用它会产生错误。

this关键字

除了使用第9章的base关键字外，还可以使用this关键字。与base一样，this也可以用在类成员的内部，且该关键字也引用对象实例。只是this引用的是当前的对象实例（即不能在静态成员中使用this关键字，因为静态成员不是对象实例的一部分）。

this关键字最常用的功能是把当前对象实例的引用传递给一个方法，如下例所示：

```
public void doSomething()  
{
```



```
MyTargetClass myObj = new MyTargetClass();  
myObj.DoSomethingWith(this);  
}
```

其中，被实例化的MyTargetClass实例（myObj）有一个DoSomethingWith()方法，该方法带一个参数，其类型与包含上述方法的类兼容。这个参数类型可以是类的类型、由这个类继承的类类型，或者由这个类或System.Object实现的一个接口。

this关键字的另一个常见用法是限定局部类型的成员，例如：

```
public class MyClass  
{  
    private int someData;  
    public int SomeData  
    {  
        get  
        {  
            return this  
                .someData;  
        }  
    }  
}
```

许多开发人员都喜欢这个语法，它可以用于任意成员类型，因为可以一眼看出引用的是成员，而不是局部变量。

10.2.3 嵌套的类型定义

除了在名称空间中定义类型（如类）之外，还可以在其他类中定义它们。如果这么做，就可以在定义中使用各种访问修饰符，而不仅是 `public` 和 `internal`，也可以使用 `new` 关键字来隐藏继承于基类的类型定义。例如，以下代码定义了 `MyClass`，也定义了一个嵌套的类 `myNestedClass`：

```
public class MyClass
{
    public class MyNestedClass

{

    public int NestedClassField;

}
```

```
}
```

如果要在MyClass的外部实例化myNestedClass，就必须限定名称，例如：

```
MyClass.MyNestedClass myObj = new MyClass.MyNestedClass();
```

但是，如果嵌套的类声明为私有，就不能这么做。这个功能主要用来定义对于其包含类来说是私有的类，这样，名称空间中的其他代码就不能访问它。使用该功能的另一个原因是嵌套类可以访问其包含类的私有和受保护成员。接下来的示例演示了嵌套类。

试一试：使用嵌套类： **Ch10Ex02**

（1）在C:\BegVCSharp\Chapter10目录中创建一个新的控制台应用程序Ch10Ex02。

（2）修改Program.cs中的代码，如下所示：

```
namespace Ch10Ex02
{
    public class ClassA

    {
```

```
private int state = -1;
```

```
public int State
```

```
{
```

```
    get { return state; }
```

```
}
```

```
public class ClassB
```

```
{
```

```
public void SetPrivateState(ClassA target, int newState

{

    target.state = newState;

}

}

}
```

```
class Program
{
    static void Main(string[] args)
    {
        ClassA myObject = new ClassA();

        WriteLine($"myObject.State = {myObject.State}");

        ClassA.ClassB myOtherObject = new ClassA.ClassB();

        myOtherObject.SetPrivateState(myObject, 999);

        WriteLine($"myObject.State = {myObject.State}");

        ReadKey();
    }
}
```

```
}  
}  
}
```

(3) 运行应用程序，结果如图10-3所示。

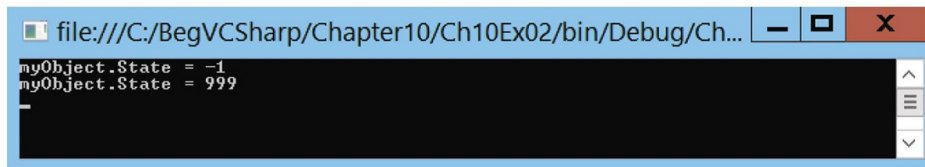


图10-3

示例的说明

Main()中的代码创建并使用了ClassA的一个实例，该类包含一个只读属性State。然后创建了嵌套类ClassA.ClassB的一个实例。该嵌套类能够访问ClassA.State的底层字段ClassA.state，即使这个字段是一个私有字段。因此，嵌套类的方法SetPrivateState()可以修改ClassA的只读属性State的值。

有必要再次重申一下，之所以可以这么做，是因为ClassB被定义为ClassA的嵌套类。如果把ClassB的定义移出ClassA，那么上面的代码就会产生如下编译错误：

```
'Ch10Ex02.ClassA.state' is inaccessible due to its protection
```

将类的内部状态提供给它嵌套类这一功能在某些情况下十分有用。但是，大多数时候通过类提供的方法操作其内部状态就足够了。

10.3 接口的实现

在继续前，先讨论一下如何定义和实现接口。第9章介绍过接口的定义方式与类相似，使用的代码如下：

```
interface IMyInterface
{
    // Interface members.
}
```

接口成员的定义与类成员的定义相似，但具有几个重要的区别：

- 不允许使用访问修饰符（`public`、`private`、`protected`或`internal`），所有接口成员都是隐式公共的。
- 接口成员不能包含代码体。
- 接口不能定义字段成员。
- 不能用关键字`static`、`virtual`、`abstract`或`sealed`来定义接口成员。
- 类型定义成员是禁止的。

但要隐藏从基接口中继承的成员，可以用关键字`new`来定义它们，例如：

```
interface IMyBaseInterface
{
    void DoSomething();
}
```



```
interface IMyDerivedInterface : IMyBaseInterface
{
    new void DoSomething();
}
```

其方式与隐藏继承的类成员的方式一样。

在接口中定义的属性可以定义访问块`get`和`set`中的哪一个能用于该属性（或将它们同时用于该属性），例如：

```
interface IMyInterface
{
    int MyInt { get; set; }
}
```

其中`int`属性`MyInt`有`get`和`set`存取器。对于访问级别有更严格限制的属性来说，可以省略它们中的任一个。

注意： 这个语法类似于自动属性，但自动属性是为类（而不是接口）定义的，自动属性必须包含`get`和`set`存取器。

接口没有指定应如何存储属性数据。接口不能指定字段，例如用于存储属性数据的字段。最后，接口与类一样，可以定义为类的成员（但

不能定义为其他接口的成员，因为接口不能包含类型定义）。

在类中实现接口

实现接口的类必须包含该接口所有成员的实现代码，且必须匹配指定的签名（包括匹配指定的`get`和`set`块），并且必须是公共的。例如：

```
public interface IMyInterface
{
    void DoSomething();

    void DoSomethingElse();
}

public class MyClass : IMyInterface
{
    public void DoSomething() {}

    public void DoSomethingElse() {}
}
```

```
}
```

可使用关键字**virtual**或**abstract**来实现接口成员，但不能使用**static**或**const**。还可以在基类上实现接口成员，例如：

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}
public class MyBaseClass
{
    public void DoSomething()
}
{}
}
public class MyDerivedClass : MyBaseClass, IMyInterface
{
    public void DoSomethingElse() {}
}
```

继承一个实现给定接口的基类，就意味着派生类隐式地支持这个接口，例如：

```
public interface IMyInterface
{
```

```

        void DoSomething();
        void DoSomethingElse();
    }
    public class MyBaseClass : IMyInterface
    {
        public virtual void DoSomething() {}
        public virtual void DoSomethingElse() {}
    }
    public class MyDerivedClass : MyBaseClass
    {
        public override void DoSomething()

    {}
    }

```

显然，在基类中把实现代码定义为虚拟，派生类就可以替换该实现代码，而不是隐藏它们。如果要使用new关键字隐藏一个基类成员，而不是重写它，则方法IMyInterface. DoSomething()就总是引用基类版本，即使通过这个接口来访问派生类，也是这样。

1. 显式实现接口成员

也可以由类显式地实现接口成员。如果这么做，就只能通过接口来访问该成员，不能通过类来访问。上一节的代码中使用的隐式成员可以通过类和接口来访问。

例如，如果类MyClass隐式地实现接口IMyInterface的方法DoSomething()，如上所述，则下面的代码就是有效的：

```
MyClass myObj = new MyClass();  
myObj.DoSomething();
```

下面的代码也是有效的：

```
MyClass myObj = new MyClass();  
IMyInterface myInt = myObj;  
myInt.DoSomething();
```

另外，如果MyDerivedClass显式地实现DoSomething()，就只能使用后一种技术。其代码如下：

```
public class MyClass : IMyInterface  
{  
    void IMyInterface.DoSomething() {}  
  
    public void DoSomethingElse() {}  
  
}
```

其中DoSomething()是显式实现的，而DoSomethingElse()是隐式实现的。只有后者可以直接通过MyClass的对象实例来访问。

2. 其他属性存取器

前面说过，如果实现带属性的接口，就必须实现匹配的get/set存取器。这并不是绝对正确的——如果在定义属性的接口中只包含set块，就可给类中的属性添加get块，反之亦然。但只有隐式实现接口时才能这么做。另外，大多数时候，都想让所添加的存取器的可访问修饰符比接口中定义的存取器的可访问修饰符更严格。因为按照定义，接口定义的存取器是公共的，也就是说，只能添加非公共的存取器。例如：

```
public interface IMyInterface
{
    int MyIntProperty { get

; }
}

public class MyBaseClass : IMyInterface
{
    public int MyIntProperty { get; protected set

; }
}
```

如果将新添加的存取器定义为公共的，那么能够访问实现该接口的类的代码也可以访问该存取器。但是，只能访问接口的代码就不能访问该存取器。

10.4 部分类定义

如果所创建的类包含一种类型或其他类型的许多成员时，就很容易引起混淆，代码文件也比较长。这里可以采用前面章节介绍的一种方法，即给代码分组。在代码中定义区域，就可以折叠和展开各个代码区，使代码更便于阅读。例如，有一个类的定义如下：

```
public class MyClass
{
    #region Fields
    private int myInt;
    #endregion

    #region Constructor
    public MyClass() { myInt = 99; }
    #endregion

    #region Properties
    public int MyInt
    {
        get { return myInt; }
        set { myInt = value; }
    }
    #endregion

    #region Methods
    public void DoSomething()
    {
```

```
        // Do something..  
    }  
    #endregion  
}
```

上述代码可以展开和折叠类的字段、属性、构造函数和方法，以便集中精力考虑自己感兴趣的内容。甚至可按这种方式嵌套各个区域，这样一些区域就只能在包含它们的区域被展开后才能看到。

另一种方法是使用部分类定义（**partial class definition**）。简言之，就是使用部分类定义，把类的定义放在多个文件中。例如，可将字段、属性和构造函数放在一个文件中，而把方法放在另一个文件中。为此，在包含部分类定义的每个文件中对类使用**partial**关键字即可，如下所示：

```
public partial  
  
class MyClass { ...}
```

如果使用部分类定义，**partial**关键字就必须出现在包含部分类定义的每个文件的与此相同的位置。

例如，类MainWindow中的WPF窗口将代码存储在两个文件MainWindow.xaml.cs和MainWindow.g.i.cs中（在Solution Explorer中选择Show All Files并打开obj\Debug文件夹就可以看到它们）。这样就可以重点考虑窗体的功能，不必担心代码会被自己不感兴趣的信息搅乱。

对于部分类，最后要注意的一点是：应用于部分类的接口也会应用

于整个类，也就是说，下面的两个定义：

```
public partial class MyClass : IMyInterface1 { ... }  
public partial class MyClass : IMyInterface2 { ... }
```

和

```
public class MyClass : IMyInterface1, IMyInterface2 { ... }
```

是等价的。

部分类定义可以在一个部分类定义文件或者多个部分类定义文件中包含基类。但如果基类在多个定义文件中指定，它就必须是同一个基类，因为在C#中，类只能继承一个基类。

10.5 部分方法定义

部分类也可以定义部分方法（`partial method`）。部分方法在一个部分类中定义（没有方法体），在另一个部分类中实现。在这两个部分类中，都要使用`partial`关键字。

```
public partial class MyClass
{
    partial void MyPartialMethod()

;
}
public partial class MyClass
{
    partial void MyPartialMethod()

{

// Method implementation
```

```
}
```

```
}
```

部分方法也可以是静态的，但它们总是私有的，且不能有返回值。它们使用的任何参数都不能是out参数，但可以是ref参数。部分方法也不能使用virtual、abstract、override、new、sealed和extern修饰符。

有了这些限制，就不太容易看出部分方法的作用了。实际上，部分方法的重要性体现在编译代码时，而不是使用代码时。考虑下面的代码：

```
public partial class MyClass
{
    partial void DoSomethingElse();
```

```
    public void DoSomething()
```

```
{
```

```
WriteLine("DoSomething() execution started.");
```

```
DoSomethingElse();
```

```
WriteLine("DoSomething() execution finished.");
```

```
}
```

```
}
```

```
public partial class MyClass
```

```
{
```

```
    partial void DoSomethingElse() =>
```

```
        WriteLine("DoSomethingElse() called.")
```

```
;
}
```

在第一个部分类定义中定义和调用部分方法`DoSomethingElse()`，在第二个部分类中实现它。在控制台应用程序中调用`DoSomething()`方法时，输出如下内容：

```
DoSomething() execution started.
DoSomethingElse() called.
DoSomething() execution finished.
```

如果删除第二个部分类定义，或者删除部分方法的全部实现代码（或者注释掉这部分代码），输出就如下所示：

```
DoSomething() execution started.
DoSomething() execution finished.
```

读者可能认为，调用`DoSomethingElse()`时，运行库发现该方法没有实现代码，因此会继续执行下一行代码。但实际上，编译代码时，如果代码包含一个没有实现代码的部分方法，编译器会完全删除该方法，还会删除对该方法的所有调用。执行代码时，不会检查实现代码，因为没有要检查的方法调用。这会略微提高性能。

与部分类一样，在定制自动生成的代码或设计器创建的代码时，部分方法是很有用的。设计器会声明部分方法，用户根据具体情形选择是否实现它。如果不实现它，就不会影响性能，因为在编译过的代码中并不存在该方法。

现在考虑为什么部分方法不能有返回类型。如果可以回答这个问题，就可以确保完全理解了这个主题，我们将此留作练习。

10.6 示例应用程序

为解释前面使用的一些技术，下面开发一个类模块，以便在后续章节中使用。这个类模块包含两个类：

- **Card**——表示一张标准的扑克牌，包含梅花、方块、红心和黑桃，其顺序是从A到K。
- **Deck**——表示一副完整的52张扑克牌，在扑克牌中可以按照位置访问各张牌，并可以洗牌。

再开发一个简单的客户程序，确保这个模块能正常使用，但现在还不开发完整的扑克牌游戏应用程序。

10.6.1 规划应用程序

这个应用程序的类库Ch10CardLib包含类。但在开始编写代码前，应规划一下需要的结构和类的功能。

1. **Card**类

Card类基本上是两个只读字段suit和rank的容器。把字段指定为只读的原因是“空白”的牌是没有意义的，牌在创建好后也不能修改。为此，要把默认的构造函数指定为私有，并提供另一个构造函数，使用给定的suit和rank建立一副扑克牌。

此外，Card类要重写System.Object的ToString()方法，这样才能获得人们可以理解的字符串，以表示扑克牌。为使编码简单一些，为两个字段suit和rank提供枚举。

Card类如图10-4所示。

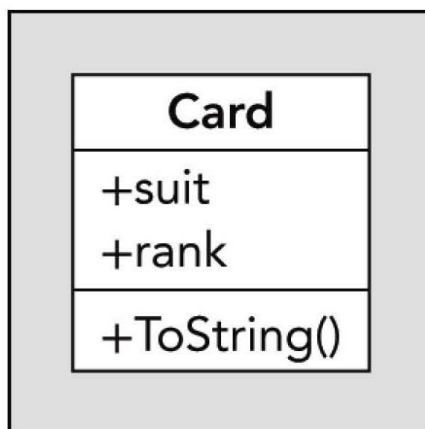


图10-4

2. Deck类

Deck类包含52个Card对象。我们为这些对象使用一个简单的数组类型。这个数组不能直接访问，因为对Card对象的访问要通过GetCard()方法来实现，该方法返回指定索引的Card对象。这个类也有一个Shuffle()方法，用于重新排列数组中的牌。Deck类如图10-5所示。

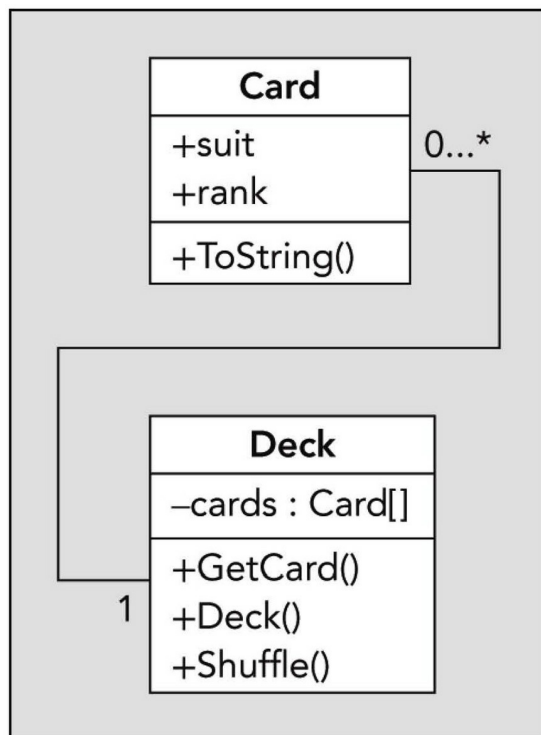


图10-5

10.6.2 编写类库

对于本例，假定读者对IDE比较熟悉，所以不再使用标准的“试一试”方式明确列出各个步骤（这些步骤已经在前面多次用过），重要的是详细讨论代码。不过，这里要包含一些提示以确保不出问题。

类和枚举都包含在一个类库项目Ch10CardLib中。这个项目将包含4个.cs文件：Card.cs包含Card类的定义，Deck.cs包含Deck类的定义，Suit.cs和Rank.cs文件包含枚举。

可使用VS的类图工具把许多代码组合在一起。

注意： 如果不愿意使用类图工具，也不必担心。下面各节都包含了类图生成的代码，所以读者完全可以理解这些内容。

首先需要完成以下操作：

（1）在C:\BegVCSharp\Chapter10目录中创建一个新类库项目Ch10CardLib。

（2）从项目中删除Class1.cs。

（3）使用Solution Explorer窗口中打开项目的类图（右击项目，然后单击View|View Class Diagram）。类图开始时应为空白，因为项目不包含类。

1. 添加**Suit**和**Rank**枚举

把一个Enum从工具箱拖动到类图中，再在显示的New Enum对话框中填写信息，就可以在类图中添加一个枚举。例如，对于Suit枚举，应在对话框中添加如图10-6所示的信息。

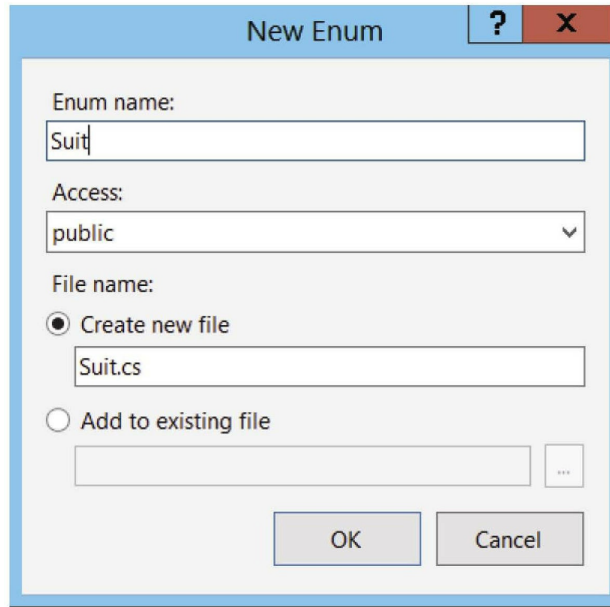


图10-6

接着使用Class Details窗口添加枚举的成员。需要添加的值如图10-7所示。



图10-7

以相同的方式利用工具箱添加Rank枚举。需要的值如图10-8所示。

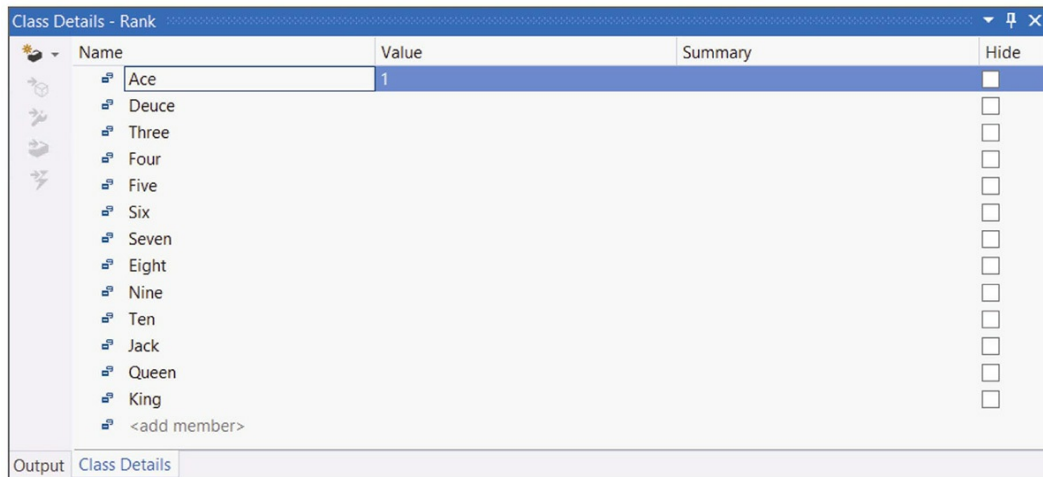


图10-8

注意： 第一个成员Ace的输入值为1，它会使枚举的底层存储匹配扑克牌的大小，例如Six就存储为6。

为这两个枚举生成的代码位于Suit.cs和Rank.cs文件中。在Ch10CardLib文件夹的Suit.cs文件中可以找到Suit枚举的完整代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Ch10CardLib
{
    public enum Suit
    {
```

```
        Club,  
        Diamond,  
        Heart,  
        Spade,  
    }  
}
```

在Ch10CardLib文件夹的Rank.cs文件中可以找到Rank枚举的完整代码，如下所示：

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
namespace Ch10CardLib  
{  
    public enum Rank  
    {  
        Ace = 1,  
        Deuce,  
        Three,  
        Four,  
        Five,  
        Six,  
        Seven,  
        Eight,  
        Nine,  
    }  
}
```

```
Ten,  
Jack,  
Queen,  
King,  
}  
}
```

另外，也可以添加Suit.cs和Rank.cs代码文件，再手工输入这些代码。注意，代码生成器在最后一个枚举成员后添加的逗号不会妨碍编译，不会创建一个额外的空成员，但它们可能会带来一些混乱。

2. 添加Card类

本节将结合使用类设计器和代码编辑器来添加Card类。使用类设计器添加类与添加枚举十分类似，也是把相应的项从工具箱拖动到类图中。这里要把Class拖动到类图中，并把新类命名为Card。

使用Class Details窗口添加字段rank和suit，再使用Properties窗口把字段的Constant Kind设置为readonly。还需要添加两个构造函数，一个是默认构造函数（私有），另一个构造函数（公共）带有两个参数：newSuit和newRank，其类型分别是Suit和Rank。最后重写ToString()，这需要在Properties窗口中修改Inheritance Modifier，将它设置为override。

图10-9显示了Class Details窗口和已输入所有信息的Card类（可在Ch10CardLib\Card.cs中找到其代码）。

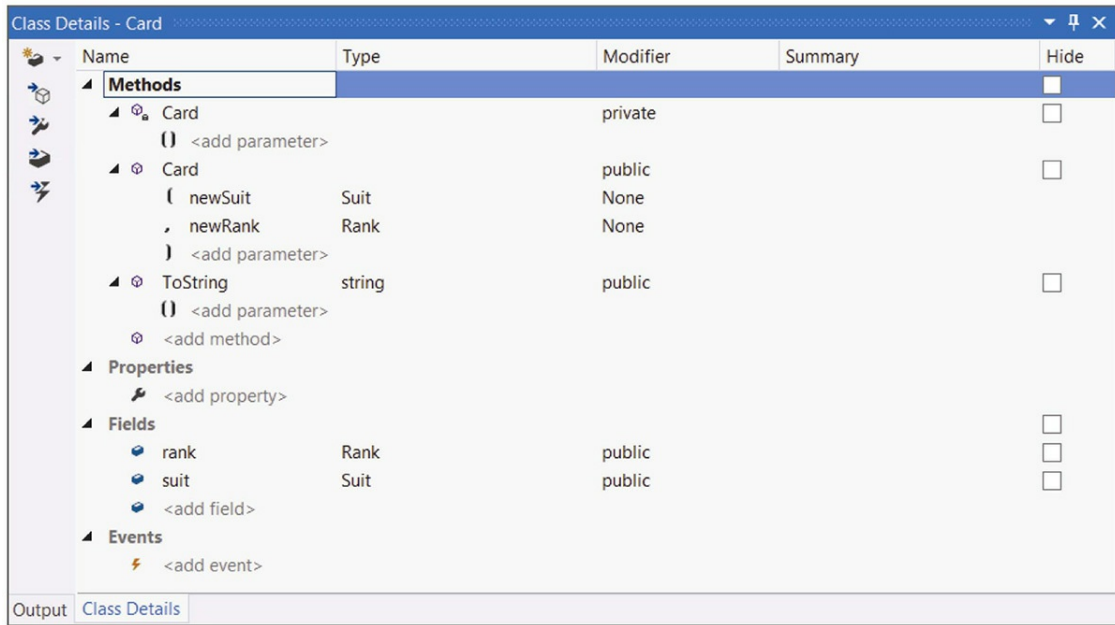


图10-9

然后需要修改Card.cs中类的代码（或者把这些代码添加到名称空间Ch10CardLib的新类Card中），如下所示：

```
public class Card
{
    public readonly Suit suit;
    public readonly Rank rank;
    public Card(Suit newSuit, Rank newRank)
    {
        suit = newSuit;

        rank = newRank;
    }
}
```

```
    }  
    private Card()  
    {  
    }  
    public override string ToString()  
    {  
        return "The " + rank + " of " + suit + "s";  
    }  
}
```

重写的ToString()方法将已存储的枚举值的字符串表示写入到返回的字符串中，非默认的构造函数初始化suit和rank字段的值。

3. 添加**Deck**类

Deck类需要使用类图定义以下成员：

- Card[]类型的私有字段cards。
- 公共的默认构造函数。
- 公共方法GetCard()，它带有一个int参数cardNum，并返回一个Card类型的对象。

- 公共方法Shuffle(), 它不带参数, 返回void。

添加这些成员后, Deck类的Class Details窗口就如图10-10所示。

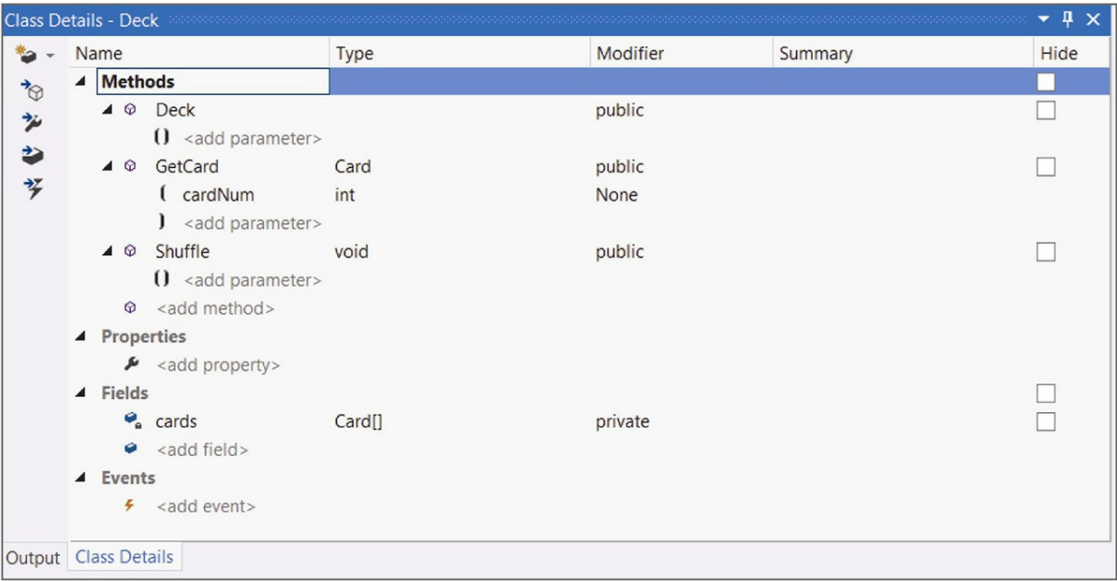


图10-10

为使类图更加清晰, 可以显示所添加的成员和类型之间的关系。在类图中依次右击下面的项, 从菜单中选择Show as Association选项:

- Deck中的cards
- Card中的suit
- Card中的rank

完成后的类图如图10-11所示。

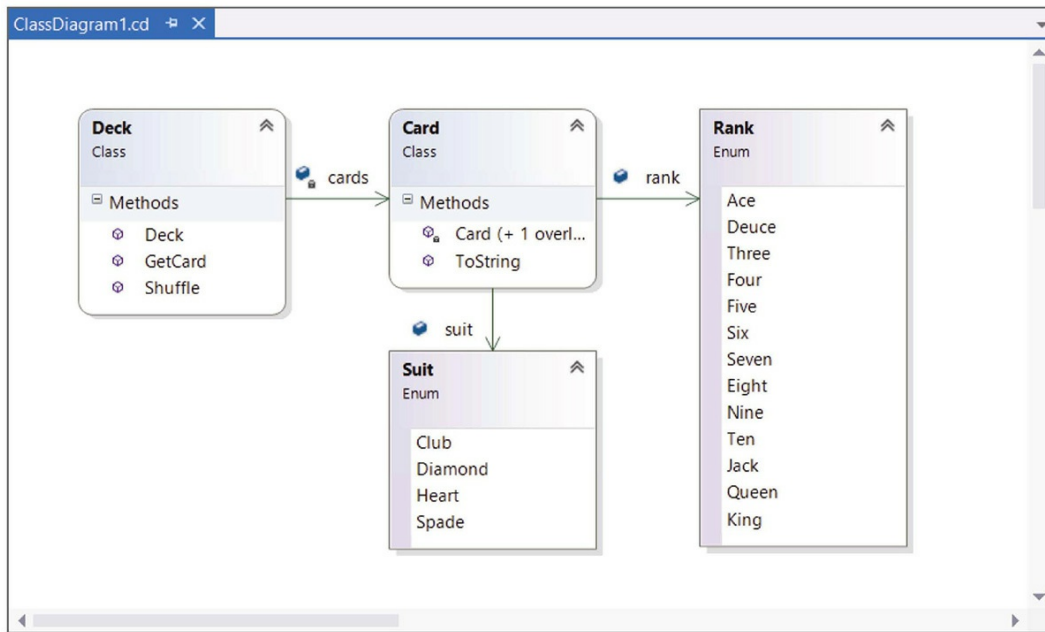


图10-11

接着修改Deck.cs中的代码（如果不使用类设计器，就必须首先使用下面的代码添加这个类）。这些代码包含在Ch10CardLib\Deck.cs中。首先实现构造函数，它在cards字段中创建52张牌，并给它们赋值。对两个枚举的所有组合进行迭代，每次迭代都创建一张牌。这将使cards最初包含一个有序的扑克牌列表：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Ch10CardLib
{
    public class Deck
```

```

{
    private Card[] cards;
    public Deck()
    {
        cards = new Card[52];

        for (int suitVal = 0; suitVal<4; suitVal++)

        {

            for (int rankVal = 1; rankVal<14; rankVal++)

            {

                cards[suitVal * 13 + rankVal -1] = new Card((Suit)su

```

(Rank)ran

}

}

}

然后实现GetCard()方法，为指定的索引返回Card对象，或者以与前面相同的方式抛出一个异常：

```
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum<= 51)

        return cards[cardNum];
```

```
else
```

```
throw (new System.ArgumentOutOfRangeException("cardNu
```

```
"Value must be between 0 and 51."));
```

```
}
```

最后实现Shuffle()方法。这个方法创建一个临时的扑克牌数组，并把扑克牌从现有的cards数组随机复制到这个数组中。这个函数的主体是一个从0~51的循环，在每次循环时，都会使用.NET Framework中System.Random类的实例生成一个0~51之间的随机数。进行实例化后，这个类的对象使用方法Next（X）生成一个介于0~X之间的随机数。有了一个随机数后，就可以将它用作临时数组中Card对象的索引，以便复制cards数组中的扑克牌。

为记录已赋值的扑克牌，我们还有一个bool变量的数组，在复制每张牌时，把该数组中的值指定为true。在生成随机数时，检查这个数组，看看是否已经把一张牌复制到临时数组中由随机数指定的位置上了，如果已经复制，将生成另一个随机数。

这不是完成该任务的最高效方式，因为生成的许多随机数都可能找不到空位置以复制扑克牌。但它仍能完成任务，而且很简单，因为C#代码的执行速度很快，我们几乎觉察不到延迟。代码如下：

```
public void Shuffle()
{
    Card[] newDeck = new Card[52];

    bool[] assigned = new bool[52];

    Random sourceGen = new Random();

    for (int i = 0; i<52; i++)

    {

        int destCard = 0;
```

```
bool foundCard = false;
```

```
while (foundCard == false)
```

```
{
```

```
    destCard = sourceGen.Next(52);
```

```
    if (assigned[destCard] == false)
```

```
        foundCard = true;
```

```
}
```

```
    assigned[destCard] = true;
```

```
    newDeck[destCard] = cards[i];
```

```
}
```

```
newDeck.CopyTo(cards, 0);
```

```
}
```

```
}
```

```
}
```

这个方法的最后一行使用System.Array类的CopyTo()方法（在创建数组时使用），把newDeck中的每张扑克牌复制回cards中。也就是说，我们使用同一个cards对象中的同一组Card对象，而不是创建新实例。如

果改用`cards=newDeck`，就会用另一个对象替代`cards`引用的对象实例。如果其他地方的代码仍保留对原`cards`实例的引用，就会出问题——不会洗牌。

至此，就完成了类库代码。

10.6.3 类库的客户应用程序

为简单起见，可以在包含类库的解决方案中添加一个客户控制台应用程序。为此，只需在Solution Explorer窗口中右击解决方案，选择Add|New Project，新项目命名为Ch10CardClient。

为在这个新的控制台应用程序项目中使用前面创建的类库，只需要添加一个对类库项目Ch10CardLib的引用。为此，可以使用Reference Manager对话框的Projects选项卡，如图10-12所示。

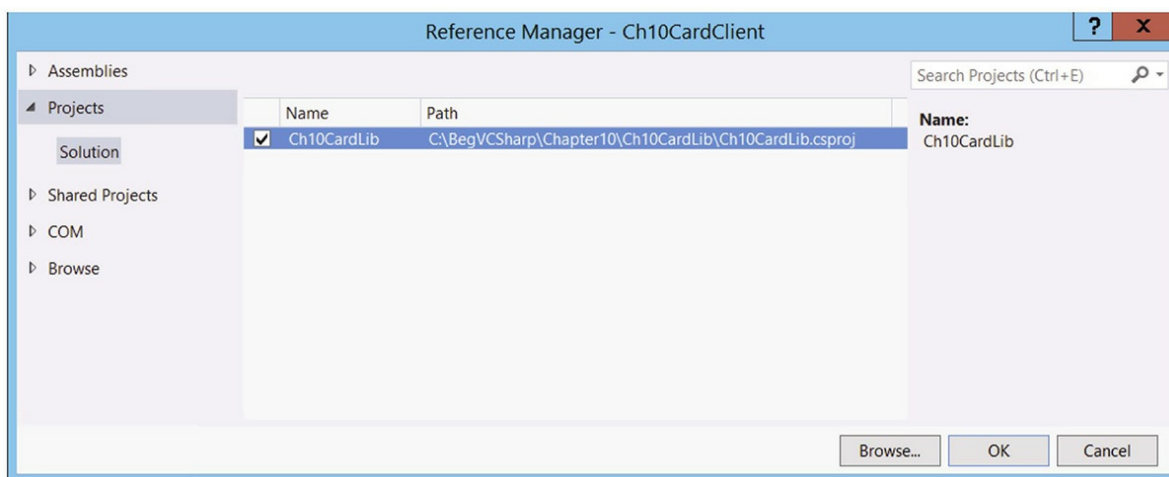


图10-12

选择项目，单击OK按钮，就添加了引用。

因为这个新项目是创建的第二个项目，所以还需要指定该项目是解决方案的启动项目，即在单击**Run**后，将执行这个项目。为此，在 **Solution Explorer**窗口中右击该项目名，选择**Set as StartUp Project**菜单项。

然后需要添加使用新类的代码，这些代码不需要做什么特别的任务，所以添加下面的代码就可以（这些代码包括在代码文件 **Ch10CardClient\Program.cs**中）：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;
using Ch10CardLib;

namespace Ch10CardClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck myDeck = new Deck();
        }
    }
}
```

```
myDeck.Shuffle();
```

```
for (int i = 0; i<52; i++)
```

```
{
```

```
    Card tempCard = myDeck.GetCard(i);
```

```
    Write(tempCard.ToString());
```

```
    if (i != 51)
```

```
        Write(", ");
```

```
else
```

```
WriteLine();
```

```
}
```

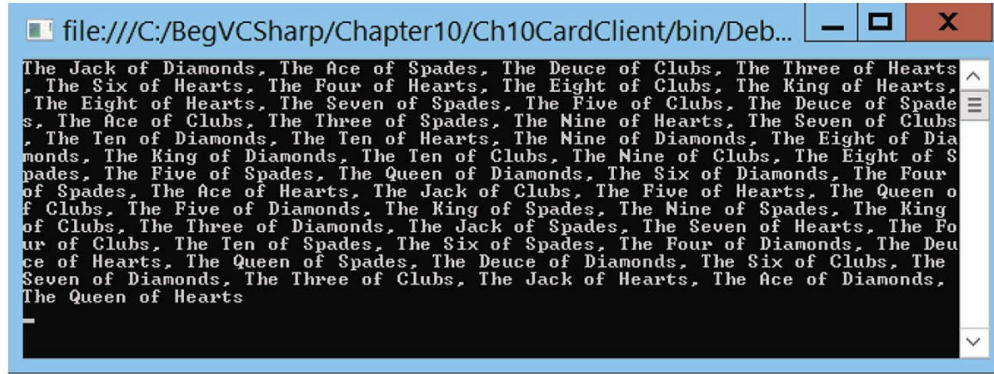
```
ReadKey();
```

```
}
```

```
}
```

```
}
```

运行应用程序的结果如图10-13所示。



```
file:///C:/BegVCSharp/Chapter10/Ch10CardClient/bin/Deb...
The Jack of Diamonds, The Ace of Spades, The Deuce of Clubs, The Three of Hearts
, The Six of Hearts, The Four of Hearts, The Eight of Clubs, The King of Hearts,
The Eight of Hearts, The Seven of Spades, The Five of Clubs, The Deuce of Spade
s, The Ace of Clubs, The Three of Spades, The Nine of Hearts, The Seven of Clubs
, The Ten of Diamonds, The Ten of Hearts, The Nine of Diamonds, The Eight of Dia
monds, The King of Diamonds, The Ten of Clubs, The Nine of Clubs, The Eight of S
pades, The Five of Spades, The Queen of Diamonds, The Six of Diamonds, The Four
of Spades, The Ace of Hearts, The Jack of Clubs, The Five of Hearts, The Queen o
f Clubs, The Five of Diamonds, The King of Spades, The Nine of Spades, The King
of Clubs, The Three of Diamonds, The Jack of Spades, The Seven of Hearts, The Fo
ur of Clubs, The Ten of Spades, The Six of Spades, The Four of Diamonds, The Deu
ce of Hearts, The Queen of Spades, The Deuce of Diamonds, The Six of Clubs, The
Seven of Diamonds, The Three of Clubs, The Jack of Hearts, The Ace of Diamonds,
The Queen of Hearts
```

图10-13

52张扑克牌是随机放置的。后续章节将继续开发和使用这个类库。

10.7 Call Hierarchy窗口

现在分析VS的另一项功能：Call Hierarchy窗口，它可以审查代码，确定方法在哪里调用，以及它们与其他方法的关系。说明这个功能的最好方式是列举一个例子。

打开上一节的示例应用程序，再打开Deck.cs代码文件，找到Shuffle()方法，右击它，选择View Call Hierarchy菜单项，将显示如图10-14所示的窗口（其中展开了一些区域）。

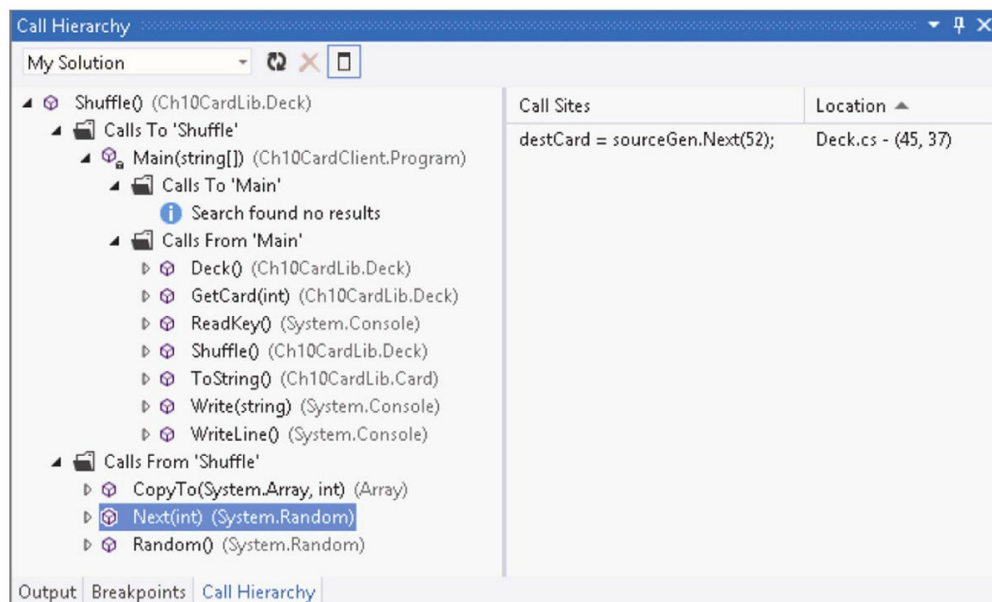


图10-14

从Shuffle()方法开始，可在窗口的树形视图中找出调用该方法的所有代码，以及这个方法进行的所有调用。例如，在Shuffle()中调用了图中突出显示的Next(int)方法，所以它显示在Calls From 'Shuffle'部分。

单击一个调用时，会在右边看到进行这个调用的代码行及其位置。双击该位置，会立即跳到进行这个调用的代码行上。

还可以沿着层次结构向下研究其中的方法，在图10-14中就是Main()方法，图中显示了从Main()方法中调用的方法和调用Main()的方法。

调试和重构代码时，这个窗口是非常有用的，因为它允许查看不同部分的代码是如何相关的。

10.8 练习

(1) 编写代码，定义一个基类MyClass，其中包含虚拟方法GetString()。这个方法应返回存储在受保护字段myString中的字符串，该字段可以通过只写公共属性ContainedString来访问。

(2) 从类MyClass中派生一个类MyDerivedClass。重写GetString()方法，使用该方法的基类实现代码从基类中返回一个字符串，但在返回的字符串中添加文本“(output from derived class)”。

(3) 部分方法定义必须使用void返回类型。说明其原因。

(4) 编写一个类MyCopyableClass，该类可以使用方法GetCopy()返回它本身的一个副本。这个方法应使用派生于System.Object的MemberwiseClone()方法。为该类添加一个简单属性，并且编写客户代码，客户代码使用该属性检查任务是否成功执行。

(5) 为Ch10CardLib库编写一个控制台客户程序，从洗牌后的Deck对象中一次取出5张牌。如果这5张牌都是相同的花色，客户程序就应在屏幕上显示这5张牌，以及文本“Flush!”，否则在取出50张牌以后就输出文本“No flush”，并退出。

附录A给出了练习答案。

10.9 本章要点

主题	要点
成员定义	可在类中定义字段、方法和属性成员。字段用可访问性、名称和类型定义，方法用可访问性、返回类型、名称和参数定义，属性用可访问性、名称、 <code>get</code> 和/或 <code>set</code> 存取器定义。各个属性存取器可以有自己的可访问性，但它必须低于整个属性的可访问性
成员隐藏和重写	属性和方法可在基类中定义为抽象或虚拟，以定义继承。派生类必须实现抽象的成员，使用 <code>override</code> 关键字可以重写虚拟的成员。派生类还可以用 <code>new</code> 关键字提供新的实现代码，用 <code>sealed</code> 关键字禁止进一步重写虚拟成员。可用 <code>base</code> 关键字调用基类的实现代码
接口的实现	实现了接口的类必须实现该接口定义为公共的所有成员。可以隐式或显式实现接口，其中显式实现代码只能通过接口引用来使用
部分定义	使用 <code>partial</code> 关键字可以把类定义放在多个代码文件中。还可以使用 <code>partial</code> 关键字创建部分方法。部分方法有一些限制，包括没有返回值或 <code>out</code> 参数，如果没有提供实现代码，就不能编译部分方法

第11章 集合、比较和转换

本章内容：

- 如何定义和使用集合
- 可以使用的不同类型的集合
- 如何比较类型，如何使用is运算符
- 如何比较值，如何重载运算符
- 如何定义和使用转换
- 如何使用as运算符

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 11 Code后，可以找到与本章示例对应的单独文件。

前面讨论了C#中所有的基本OOP技术，读者还应熟悉一些比较高级的技术。在编写代码时，经常需要使用这些技术解决某些问题。学习这些技术可以让开发过程更加顺畅，让你把注意力集中到应用程序其他更重要的方面。本章主要内容如下：

- 集合： 可以使用集合来维护对象组。与前面章节使用的数组不

同，集合可以包含更高级的功能，例如，控制对它们包含的对象的访问、搜索和排序等。本章将介绍如何使用和创建集合类，学习充分利用它们的一些强大技术。

- 比较： 在处理对象时，常要比较它们。这对于集合尤其重要，因为这是排序的实现方式。本章将介绍如何以各种方式比较对象（包括运算符重载），如何使用`Comparable`和`Comparer`接口对集合排序。
- 转换： 前面的章节介绍了如何把对象从一种类型转换为另一种类型。本章讨论如何定制类型转换，以满足自己的需要。

11.1 集合

第5章介绍了如何使用数组创建包含许多对象或值的变量类型。但数组有一定的限制。最大的限制是一旦创建好数组，它们的大小就是固定的，不能在现有数组的末尾添加新项，除非创建一个新的数组。这常常意味着用于处理数组的语法比较复杂。OOP技术可以创建在内部执行大多数此类处理的类，因此简化了使用项列表或数组的代码。

C#中的数组实现为System.Array类的实例，它们只是集合类（Collection Class）中的一种类型。集合类一般用于处理对象列表，其功能比简单数组要多，功能大多是通过实现System.Collections名称空间中的接口而获得的，因此集合的语法已经标准化了。这个名称空间还包含其他一些有趣的东西，例如，以不同于System.Array的方式实现这些接口的类。

集合的功能（包括基本功能，例如，用[index]语法访问集合中的项）可以通过接口来实现，所以不仅可以使基本集合类，例如System.Array，还可以创建自己的定制集合类。这些集合可以专用于要枚举的对象（即要从中建立集合的对象）。这么做的一个优点是定制的集合类可以是强类型化的。也就是说，从集合中提取项时，不需要把它们转换为正确类型。另一个优点是提供专用的方法，例如，可以提供获得项子集的快捷方法。在扑克牌示例中，可以添加一个方法，来获得特定花色中的所有Card项。

System.Collections名称空间中的几个接口提供了基本的集合功能：

- IEnumerable可以迭代集合中的项。
- ICollection（继承于IEnumerable）可以获取集合中项的个数，并能把项复制到一个简单的数组类型中。
- IList（继承于IEnumerable和ICollection）提供了集合的项列表，允许访问这些项，并提供其他一些与项列表相关的基本功能。
- IDictionary（继承于IEnumerable和ICollection）类似于IList，但提供了可通过键值（而不是索引）访问的项列表。

System.Array类实现了IList、ICollection和IEnumerable，但不支持IList的一些更高级功能，它表示大小固定的项列表。

11.1.1 使用集合

Systems.Collections名称空间中的类System.Collections.ArrayList也实现了IList、ICollection和IEnumerable接口，但实现方式比System.Array更复杂。数组的大小是固定不变的（不能添加或删除元素），而这个类可以用于表示大小可变的项列表。为了更准确地理解这个高级集合的功能，下面列举一个使用这个类和一个简单数组的示例。

试一试：数组和高级集合：**Ch11Ex01**

（1）在C:\BegVCSharp\Chapter11目录中创建一个新的控制台应用程序Ch11Ex01。

（2）在Solution Explorer窗口中右击项目，选择Add|Class选项，给

项目添加3个新类：Animal、Cow和Chicken。

(3) 修改Animal.cs中的代码，如下所示：

```
namespace Ch11Ex01
{
    public abstract class Animal

    {

        protected string name;

        public string Name

        {

            get { return name; }
        }
    }
}
```

```
set { name = value; }
```

```
}
```

```
public Animal()
```

```
{
```

```
name = "The animal with no name";
```

```
}
```

```
public Animal(string newName)
```

```
{
```

```
    name = newName;
```

```
}
```

```
public void Feed() => WriteLine($"{name} has been fed."),
```

```
}
```

```
}
```

(4) 修改Cow.cs中的代码，如下所示：

```
namespace Ch11Ex01
```

```
{
```

```
public class Cow : Animal
```

```
{
```

```
    public void Milk() => WriteLine($"{name} has been milked
```

```
    public Cow(string newName) : base(newName) {}
```

```
}
```

```
}
```

（5）修改Chicken.cs中的代码，如下所示：

```
namespace Ch11Ex01
```

```
{
```

```
    public class Chicken : Animal
```



```
{
```

```
public void LayEgg() => WriteLine($"{name} has laid an e
```

```
public Chicken(string newName) : base(newName) {}
```

```
}
```

```
}
```

（6）修改Program.cs中的代码，如下所示：

```
using System;
```

```
using System.Collections;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

```
using System.Threading.Tasks;
```

```
using static System.Console;
```

```
namespace Ch11Ex01
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Create an Array type collection of Animal "

                    "objects and use it:");

            Animal[] animalArray = new Animal[2];

            Cow myCow1 = new Cow("Lea");

            animalArray[0] = myCow1;

            animalArray[1] = new Chicken("Noa");
```

```
foreach (Animal myAnimal in animalArray)
```

```
{
```

```
    WriteLine($"New {myAnimal.ToString()} object added to
```

```
        $" collection, Name = {myAnimal.Name}");
```

```
}
```

```
WriteLine($"Array collection contains {animalArray.Length}
```

```
animalArray[0].Feed();
```

```
((Chicken)animalArray[1]).LayEgg();
```

```
WriteLine();
```

```
WriteLine("Create an ArrayList type collection of Anima
```

```
"objects and use it:");
```

```
ArrayList animalArrayList = new ArrayList();
```

```
Cow myCow2 = new Cow("Rual");
```

```
animalArrayList.Add(myCow2);
```

```
animalArrayList.Add(new Chicken("Andrea"));
```

```
foreach (Animal myAnimal in animalArrayList)
```

```
{
```

```
    WriteLine($"New {myAnimal.ToString()} object added to
```

```
        $" collection, Name = {myAnimal.Name}");
```

```
}
```

```
WriteLine($"ArrayList collection contains {animalArrayL
```

```
+ "objects.");
```

```
((Animal)animalArrayList[0]).Feed();
```

```
((Chicken)animalArrayList[1]).LayEgg();
```

```
WriteLine();
```

```
WriteLine("Additional manipulation of ArrayList:");
```

```
animalArrayList.RemoveAt(0);
```

```
((Animal)animalArrayList[0]).Feed();
```

```
animalArrayList.AddRange(animalArray);
```

```
((Chicken)animalArrayList[2]).LayEgg();
```

```
WriteLine($"The animal called {myCow1.Name} is at " +
```

```
    $"index {animalArrayList.IndexOf(myCow1)}.");
```

```
myCow1.Name = "Mary";
```

```
WriteLine("The animal is now " +
```

```
$" called {((Animal)animalArrayList[1]).Name };
```

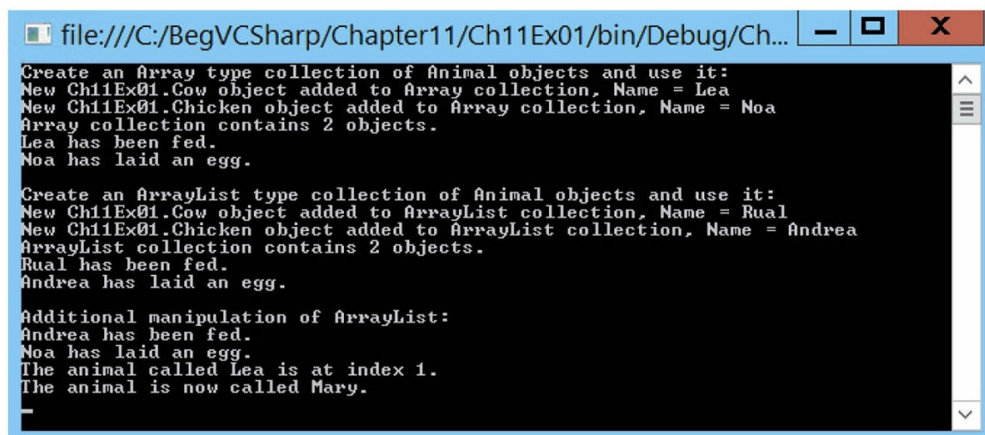
```
ReadKey();
```

```
}
```

```
}
```

```
}
```

(7) 运行该应用程序，其结果如图11-1所示。



```
file:///C:/BegVCSharp/Chapter11/Ch11Ex01/bin/Debug/Ch...
Create an Array type collection of Animal objects and use it:
New Ch11Ex01.Cow object added to Array collection, Name = Lea
New Ch11Ex01.Chicken object added to Array collection, Name = Noa
Array collection contains 2 objects.
Lea has been fed.
Noa has laid an egg.

Create an ArrayList type collection of Animal objects and use it:
New Ch11Ex01.Cow object added to ArrayList collection, Name = Rual
New Ch11Ex01.Chicken object added to ArrayList collection, Name = Andrea
ArrayList collection contains 2 objects.
Rual has been fed.
Andrea has laid an egg.

Additional manipulation of ArrayList:
Andrea has been fed.
Noa has laid an egg.
The animal called Lea is at index 1.
The animal is now called Mary.
```

图11-1

示例的说明

这个示例创建了两个对象集合，第一个集合使用`System.Array`类（这是一个简单数组），第二个集合使用`System.Collections.ArrayList`类。这两个集合都是`Animal`对象，在`Animal.cs`中定义。`Animal`类是抽象类，所以不能进行实例化。但通过多态性（详见第8章），可使集合中的项成为派生于`Animal`类的`Cow`和`Chicken`类实例。

在`Class1.cs`的`Main()`方法中创建好这些数组后，就可以显示其特性和功能。有几个处理操作可以应用到`Array`和`ArrayList`集合上，但它们的语法略有区别。也有一些操作只能使用更高级的`ArrayList`类型。

下面首先通过比较这两种集合类型的代码和结果，讨论一下类似操作。首先是集合的创建。对于简单数组而言，只有用固定的大小来初始化数组，才能使用它。下面使用第5章介绍的标准语法来创建数组
`animalArray`:

```
Animal[] animalArray = new Animal[2];
```

而`ArrayList`集合不需要初始化其大小，所以可使用以下代码创建
`animalArrayList`列表:

```
ArrayList animalArrayList = new ArrayList();
```

这个类还有另外两个构造函数。第一个构造函数把现有的集合作为一个参数，将其内容复制到新实例中；而另一个构造函数通过一个参数设置集合的容量（`capacity`）。这个容量用一个`int`值指定，设置集合中可以包含的初始项数。但这并不是绝对容量，因为如果集合中的项数超过了这个值，容量就会自动增加一倍。

因为数组是引用类型（例如，`Animal`和`Animal`派生的对象），所以用一个长度初始化数组并没有初始化它所包含的项。要使用一个指定的项，该项还需要初始化，即需要给这个项赋予初始化了的对象：

```
Cow myCow1 = new Cow("Lea");  
animalArray[0] = myCow1;  
animalArray[1] = new Chicken("Noa");
```

这段代码以两种方式完成该初始化任务：用现有的`Cow`对象来赋值，或者通过创建一个新的`Chicken`对象来赋值。主要区别在于前者引用了数组中的对象——我们在代码的后面就使用了这种方式。

对于`ArrayList`集合，它没有现成的项，也没有`null`引用的项。这样就不能以相同的方式给索引赋予新实例。我们使用`ArrayList`对象的`Add()`方法添加新项：

```
Cow myCow2 = new Cow("Rual");  
animalArrayList.Add(myCow2);  
animalArrayList.Add(new Chicken("Andrea"));
```

除语法稍有不同外，还可以采用相同的方式把新对象或现有对象添加到集合中。以这种方式添加完项后，就可以使用与数组相同的语法来改写它们，例如：

```
animalArrayList[0] = new Cow("Alma");
```

但不能在这个示例中这么做。

第5章介绍了如何使用`foreach`结构迭代一个数组。这是可以的，因为`System.Array`类实现了`IEnumerable`接口，这个接口的唯一方法

GetEnumerator()可以迭代集合中的各项。后面将更深入地讨论这一点。在代码中，我们写出了数组中每个Animal对象的信息：

```
foreach (Animal myAnimal in animalArray)
{
    WriteLine($"New {myAnimal.ToString()} object added to Array
               $"collection, Name = {myAnimal.Name}");
}
```

这里使用的ArrayList对象也支持IEnumerable接口，并可以与foreach一起使用，此时语法是相同的：

```
foreach (Animal myAnimal in animalArrayList)
{
    WriteLine($"New {myAnimal.ToString()} object added to Array
               $"collection, Name = {myAnimal.Name}");
}
```

接着使用数组的Length属性，在屏幕上输出数组中元素的个数：

```
WriteLine($"Array collection contains {animalArray.Length} obj
```

也可以使用ArrayList集合得到相同的结果，但要使用Count属性，该属性是ICollection接口的一部分：

```
WriteLine($"ArrayList collection contains {animalArrayList.Cou
```

如果不能访问集合——无论是简单数组，还是较复杂的集合——中的项，它们就没什么用途。简单数组是强类型化的，可以直接访问它们所包含的项类型。所以可以直接调用项的方法：

```
animalArray[0].Feed();
```

数组的类型是抽象类型`Animal`，因此不能直接调用由派生类提供的方法，而必须使用数据类型转换：

```
((Chicken)animalArray[1]).LayEgg();
```

`ArrayList`集合是`System.Object`对象的集合（通过多态性赋给`Animal`对象），所以必须对所有的项进行数据类型转换：

```
((Animal)animalArrayList[0]).Feed();  
((Chicken)animalArrayList[1]).LayEgg();
```

代码的剩余部分利用的一些`ArrayList`集合功能超出了`Array`集合的功能范围。首先，可以使用`Remove()`和`RemoveAt()`方法删除项，这两个方法是在`ArrayList`类中实现的`ICollection`接口的一部分。它们分别根据项的引用或索引从数组中删除项。本例使用后一个方法删除列表中的第一项，即`Name`属性为`Hayley`的`Cow`对象：

```
animalArrayList.RemoveAt(0);
```

另外，还可以使用：

```
animalArrayList.Remove(myCow2);
```

因为这个对象已经有一个本地引用了，所以可以通过`Add()`添加对数组的一个现有引用，而不是创建一个新对象。无论采用哪种方式，集合中唯一剩余的项是`Chicken`对象，可以通过以下方式访问它：

```
((Animal)animalArrayList[0]).Feed();
```

当对ArrayList对象中的项进行修改，使数组中剩下N个项时，其索引范围变为0~N-1。例如，删除索引为0的项，会使其他项在数组中移动一个位置，所以应使用索引0（而非1）来访问Chicken对象。不再有索引为1的项了（因为集合中最初只有两个项），所以如果试图执行下面的代码，就会抛出异常：

```
((Animal)animalArrayList[1]).Feed();
```

ArrayList集合可以用AddRange()方法一次添加好几项。这个方法接受带有ICollection接口的任意对象，包括前面的代码所创建的animalArray数组：

```
animalArrayList.AddRange(animalArray);
```

为确定这是否有效，可以试着访问集合中的第三项，它将是animalArray中的第二项：

```
((Chicken)animalArrayList[2]).LayEgg();
```

AddRange()方法不是ArrayList提供的任何接口的一部分。这个方法专用于ArrayList类，证实了可以在集合类中执行定制操作，而不仅是前面介绍的接口要求的操作。这个类还提供了其他有趣的方法，如InsertRange()，它可以把数组对象插入到列表中的任何位置，还有用于排序和重新排序数组的方法。

最后，再回头来看看对同一个对象进行多个引用。使用IList接口中的IndexOf()方法可以看出，myCow1（最初添加到animalArray中的一个对象）现在是animalArrayList集合的一部分，它的索引如下：

```
WriteLine($"The animal called {myCow1.Name} is at index " +
```

```
($"{animalArrayList.IndexOf(myCow1)}.");
```

例如，接下来的两行代码通过对象引用重新命名了对象，并通过集合引用显示了新名称：

```
myCow1.Name = "Mary";  
WriteLine($"The animal is now called {((Animal)animalArrayList
```

11.1.2 定义集合

前面介绍了使用高级集合类能完成什么任务，下面讨论如何创建自己的强类型化的集合。一种方式是手动实现需要的方法，但这较费时间，在某些情况下也非常复杂。我们还可以从一个类中派生自己的集合，例如System.Collections.CollectionBase类，这个抽象类提供了集合类的大量实现代码。这是推荐使用的方式。

CollectionBase类有接口IEnumerable、ICollection和IList，但只提供了一些必要的实现代码，主要是IList的Clear()和RemoveAt()方法，以及ICollection的Count属性。如果要使用提供的功能，就需要自己实现其他代码。

为便于完成任务，CollectionBase提供了两个受保护的属性，它们可以访问存储的对象本身。我们可以使用List和InnerList，List可以通过IList接口访问项，InnerList则是用于存储项的ArrayList对象。

例如，存储Animal对象的集合类可以定义如下（稍后介绍一个较完整的实现代码）：

```

public class Animals : CollectionBase
{
    public void Add(Animal newAnimal)
    {
        List.Add(newAnimal);
    }
    public void Remove(Animal oldAnimal)
    {
        List.Remove(oldAnimal);
    }
    public Animals() {}
}

```

其中，Add()和Remove()方法实现为强类型化的方法，使用IList接口中用于访问项的标准Add()方法。这些方法现在只用于处理Animal类或派生于Animal的类，而前面介绍的ArrayList实现代码可处理任何对象。

CollectionBase类可以对派生的集合使用foreach语法。例如，可使用下面的代码：

```

WriteLine("Using custom collection class Animals:");
Animals animalCollection = new Animals();
animalCollection.Add(new Cow("Lea"));
foreach (Animal myAnimal in animalCollection)
{
    WriteLine($"New { myAnimal.ToString()} object added to cust
        $"collection, Name = {myAnimal.Name}");
}

```

```
}
```

但不能使用下面的代码：

```
animalCollection[0].Feed();
```

要以这种方式通过索引来访问项，就需要使用索引符。

11.1.3 索引符

索引符（`indexer`）是一种特殊类型的属性，可以把它添加到一个类中，以提供类似于数组的访问。实际上，可通过索引符提供更复杂的访问，因为我们可以用方括号语法定义和使用复杂的参数类型。它最常见的一个用法是对项实现简单的数字索引。

在Animal对象的Animals集合中添加一个索引符，如下所示：

```
public class Animals : CollectionBase
{
    ...
    public Animal this[int animalIndex]

    {
```



```

        get { return (Animal)List[animalIndex]; }

        Set { List[animalIndex] = value; }

    }

}

```

`this`关键字需要与方括号中的参数一起使用，除此以外，索引符与其他属性十分类似。这个语法是合理的，因为在访问索引符时，将使用对象名，后跟放在方括号中的索引参数（例如`MyAnimals[0]`）。

这段代码对`List`属性使用一个索引符（即在`IList`接口上，可以访问`CollectionBase`中的`ArrayList`，`ArrayList`存储了项）：

```
return (Animal)List[animalIndex];
```

这里需要进行显式数据类型转换，因为`IList.List`属性返回一个`System.Object`对象。注意，我们为这个索引符定义了一个类型。使用该索引符访问某项时，就可以得到这个类型。这种强类型化功能意味着，可以编写下述代码：

```
animalCollection[0].Feed();
```

而不是：

```
((Animal)animalCollection[0]).Feed();
```

这是强类型化的定制集合的另一个方便特性。下面扩展上一个示例，实践一下该特性。

试一试：实现**Animals**集合： **Ch11Ex02**

（1）在C:\BegVCSharp\Chapter11目录中创建一个新控制台应用程序Ch11Ex02。

（2）在Solution Explorer窗口中右击项目名，选择Add|Existing Item选项。

（3）从C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01目录中选择Animal.cs、Cow.cs和Chicken.cs文件，单击Add按钮。

（4）修改这3个文件中的名称空间声明，如下所示：

```
namespace Ch11Ex02
```

（5）添加一个新类Animals。

（6）修改Animals.cs中的代码，如下所示：

```
using System;
```

```
using System.Collections;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

```
using System.Threading.Tasks;
```

```
namespace Ch11Ex02
```

```
{
```

```
    public
```

```
class Animals : CollectionBase
```

```
{
```

```
    public void Add(Animal newAnimal)
```

```
{
```

```
        List.Add(newAnimal);
```

```
}
```

```
public void Remove(Animal newAnimal)
```

```
{
```

```
List.Remove(newAnimal);
```

```
}
```

```
public Animal this[int animalIndex]
```

```
{  
  
    get { return (Animal)List[animalIndex]; }  
  
    set { List[animalIndex] = value; }  
  
}  
  
}  
}
```

(7) 修改Program.cs, 如下所示:

```
static void Main(string[] args)  
{  
    Animals animalCollection = new Animals();
```

```
animalCollection.Add(new Cow("Donna"));
```

```
animalCollection.Add(new Chicken("Kevin"));
```

```
foreach (Animal myAnimal in animalCollection)
```

```
{
```

```
    myAnimal.Feed();
```

```
}
```

```
ReadKey();
```

}

(8) 执行应用程序，其结果如图11-2所示。

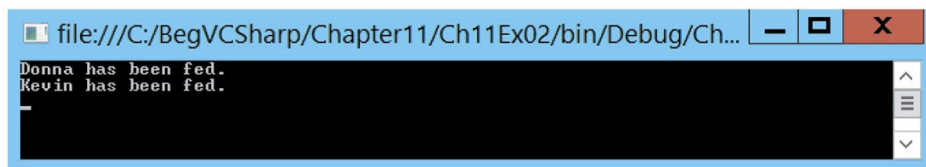


图11-2

示例的说明

这个示例使用上一节详细介绍的代码，实现Animals类中强类型化的Animal对象集合。Main()中的代码仅实例化了一个Animals对象animalCollection，添加了两个项（它们分别是Cow和Chicken的实例），再使用foreach循环调用这两个对象继承于基类Animal的Feed()方法。

11.1.4 给CardLib添加Cards集合

第10章创建了一个类库项目Ch10CardLib，它包含一个表示扑克牌的Card类和一个表示一副扑克牌的Deck类，这个Deck类是Card类的集合，且实现为一个简单数组。

本章给这个库添加一个新类，并将类库重命名为Ch11CardLib。这个新类Cards是Card对象的一个定制集合，并拥有本章前面介绍的各种功能。在C:\BegVCSharp\Chapter11目录中创建一个新的类库

Ch11CardLib。然后删除自动生成的Class1.cs文件，再使用Project|Add Existing Item，选择C:\BegVCSharp\Chapter10\Ch10CardLib\Ch10CardLib目录中的Card.cs、Deck.cs、Suit.cs和Rank.cs文件，把它们添加到项目中。与第10章介绍的这个项目的上一个版本相同，这里也不使用标准的“试一试”格式介绍这些变化。读者可在本章的下载代码中打开这个项目的版本，直接查看代码。

注意：在把源文件从Ch10CardLib复制到Ch11CardLib中时，必须修改名称空间声明，以引用Ch11CardLib。对用于测试的Ch10CardClient控制台应用程序，也要进行这个修改。

本章下载代码中的Ch11CardLib文件夹包含了对Ch11CardLib项目进行的各种扩展。因此，读者可能会注意到一些本例没有用到的代码，不过它们并不影响这里介绍的内容。很多这样的代码都被注释掉了，不过当学习相关示例时，可以取消对相应代码部分的注释。

如果要自己创建这个项目，就应添加一个新类Cards，修改Cards.cs中的代码，如下所示：

```
using System;

using System.Collections;


using System.Collections.Generic;
```



```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Ch11CardLib
{
    public

class Cards : CollectionBase

    {

        public void Add(Card newCard)

        {

            List.Add(newCard);
```

```
}
```

```
public void Remove(Card oldCard)
```

```
{
```

```
    List.Remove(oldCard);
```

```
}
```

```
public Card this[int cardIndex]
```

```
{
```

```
get { return (Card)List[cardIndex]; }
```

```
set { List[cardIndex] = value; }
```

```
}
```

```
///<summary>
```

```
///Utility method for copying card instances into another
```

```
///instance;aused in Deck.Shuffle(). This implementation
```

```
///source and target collections are the same size.
```

```
///  
///</summary>
```

```
public void CopyTo(Cards targetCards)
```

```
{
```

```
    for (int index = 0; index<this.Count; index++)
```

```
    {
```

```
        targetCards[index] = this[index];
```

```
}
```

```
}
```

```
///<summary>
```

```
/// Check to see if the Cards collection contains a part:
```

```
/// This calls the Contains() method of the ArrayList for
```

```
/// which you access through the InnerList property.
```

```
///</summary>
```

```

        public bool Contains(Card card) => InnerList.Contains(card)

    }

}

```

然后需要修改Deck.cs，以利用这个新集合（而不是数组）：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11CardLib
{
    public class Deck
    {
        private Cards cards = new Cards();
    }
}

```

```

public Deck()
{
    // Line of code removed here

    for (int suitVal = 0; suitVal<4; suitVal++)
    {
        for (int rankVal = 1; rankVal<14; rankVal++)
        {
            cards.Add(new Card((Suit)suitVal, (Rank)rankVal));

        }
    }
}

public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum<= 51)
        return cards[cardNum];
    else
        throw (new System.ArgumentOutOfRangeException("card
            "Value must be between 0 and 51."));
}

public void Shuffle()

```

```

{
    Cards newDeck = new Cards();

    bool[] assigned = new bool[52];
    Random sourceGen = new Random();
    for (int i = 0; i<52; i++)
    {
        int sourceCard = 0;

        bool foundCard = false;
        while (foundCard == false)
        {
            sourceCard = sourceGen.Next(52);

            if (assigned[sourceCard] == false)

                foundCard = true;
        }
        assigned[sourceCard] = true;
    }
}

```



```

        newDeck.Add(cards[sourceCard]);

    }

    newDeck.CopyTo(cards);

}

}

}

```

在此不需要做很多修改。其中大多数修改都涉及改变洗牌逻辑，才能把cards中随机的一张牌添加到新Cards集合newDeck的开头，而不是把cards集合中顺序位置的一张牌添加newDeck集合的随机位置上。

Ch10CardLib解决方案的客户控制台应用程序Ch10CardClient可使用这个新库得到与以前相同的结果，因为Deck的方法签名没有改变。这个类库的客户程序现在可以使用Cards集合类，而不是依赖Card对象数组，例如，在扑克牌游戏应用程序中定义一手牌。

11.1.5 键控集合和IDictionary

除IList接口外，集合还可以实现类似的IDictionary接口，允许项通过键值（如字符串名）进行索引，而不是通过一个索引。这也可以使用索引符来完成，但这次的索引符参数是与存储的项相关联的键，而不是int索引，这样集合就更便于用户使用了。

与索引的集合一样，可使用一个基类简化IDictionary接口的实现，这个基类就是DictionaryBase，它也实现IEnumerable和ICollection，提供了对任何集合都相同的基本集合处理功能。

DictionaryBase与CollectionBase一样，实现通过其支持的接口获得的一些成员（但不是全部成员）。DictionaryBase也实现Clear和Count成员，但不实现RemoveAt()。这是因为RemoveAt()是IList接口中的一个方法，不是IDictionary接口中的一个方法。但是，IDictionary有一个Remove()方法，这是一个应在基于DictionaryBase的定制集合类上实现的方法。

下面的代码是Animals类的另一个版本，这次该类派生于DictionaryBase。下面代码包括Add()、Remove()和一个通过键访问的索引符的实现代码：

```
public class Animals : DictionaryBase
{
    public void Add(string newID, Animal newAnimal)
    {
        Dictionary.Add(newID, newAnimal);
    }
    public void Remove(string animalID)
    {
```

```

        Dictionary.Remove(animalID);
    }
    public Animals() {}
    public Animal this[string animalID]
    {
        get { return (Animal)Dictionary[animalID]; }
        set { Dictionary[animalID] = value; }
    }
}

```

这些成员的区别如下：

- **Add()**——带有两个参数：一个键和一个值，存储在一起。字典集合有一个继承于DictionaryBase的成员Dictionary，这个成员是一个IDictionary接口，有自己的Add()方法，该方法带有两个object参数。我们的实现代码使用一个string值作为键，使用一个Animal对象作为与该键存储在一起的数据。
- **Remove()**——以一个键（而不是对象引用）作为参数。与指定键值对应的项被删除。
- **Indexer**——使用一个字符串键值，而不是一个索引，用于通过Dictionary的继承成员来访问存储的项，这里仍需进行数据类型转换。

基于DictionaryBase的集合和基于CollectionBase的集合之间的另一个区别是foreach的工作方式稍有区别。上一节的集合可以直接从集合中提取Animal对象。使用foreach和DictionaryBase派生类可以提供DictionaryEntry结构，这是另一个在System.Collections名称空间中定义的类型。要得到Animal对象本身，就必须使用这个结构的Value成员，

也可以使用结构的Key成员得到相关的键。要使代码等价于前面的代码：

```
foreach (Animal myAnimal in animalCollection)
{
    WriteLine($"New {myAnimal.ToString()} object added to custom collection, Name = {myAnimal.Name}");
}
```

需要使用以下代码：

```
foreach (DictionaryEntry myEntry in animalCollection)
{
    WriteLine($"New {myEntry.Value.ToString()} object added to custom collection, Name = {(Animal)myEntry.Value}");
}
```

有许多方式可以重写这段代码，以便直接通过foreach访问Animal对象，最简单的方式是实现一个迭代器。

11.1.6 迭代器

本章前面介绍过，IEnumerable接口允许使用foreach循环。在foreach循环中并不是只能使用集合类（如本章前面所示的几个集合类），相反，在foreach循环中使用定制类通常有很多优点。

但是，重写使用foreach循环的方式或者提供定制的实现方式并不一定很简单。为了说明这一点，下面深入研究foreach循环。在foreach循环

中，迭代一个collectionObject集合的过程如下：

（1）调用collectionObject.GetEnumerator()，返回一个IEnumerator引用。这个方法可以通过IEnumerable接口的实现代码来获得，但这是可选的。

（2）调用所返回的IEnumerator接口的MoveNext()方法。

（3）如果MoveNext()方法返回true，就使用IEnumerator接口的Current属性来获取对象的一个引用，用于foreach循环。

（4）重复前面两步，直到MoveNext()方法返回false为止，此时循环停止。

所以，为在类中进行这些操作，必须重写几个方法，跟踪索引，维护Current属性，以及执行其他一些操作，这要做许多工作。

一个较简单的替代方法是使用迭代器。使用迭代器将有效地自动生成许多代码，正确地完成所有任务。而且，使用迭代器的语法掌握起来非常容易。

迭代器的定义是，它是一个代码块，按顺序提供了要在foreach块中使用的所有值。一般情况下，这个代码块是一个方法，但也可以使用属性访问器和其他代码块作为迭代器。这里为简单起见，仅介绍方法。

无论代码块是什么，其返回类型都是有限制的。与期望正好相反，这个返回类型与所枚举的对象类型不同。例如，在表示Animal对象集合的类中，迭代器块的返回类型不可能是Animal。两种可能的返回类型是前面提到的接口类型IEnumerable和IEnumerator。使用这两个类型的场合是：

- 如果要迭代一个类，可使用方法GetEnumerator()，其返回类型是IEnumerator。
- 如果要迭代一个类成员，例如一个方法，则使用IEnumerable。

在迭代器块中，使用yield关键字选择要在foreach循环中使用的值。其语法如下：

```
yield return <value>  
  
>;
```

利用这个信息就足以建立一个非常简单的示例，如下所示（包含在代码文件SimpleIterators\Program.cs中）：

```
public static IEnumerable SimpleList()  
{  
    yield return "string 1";  
    yield return "string 2";  
    yield return "string 3";  
}  
static void Main(string[] args)  
{  
    foreach (string item in SimpleList())  
        WriteLine(item);  
    ReadKey();  
}
```

在此，静态方法SimpleList()就是迭代器块。它是一个方法，所以使用IEnumerable返回类型。SimpleList()使用yield关键字为使用它的foreach块提供了3个值，每个值都输出到屏幕上，结果如图11-3所示。

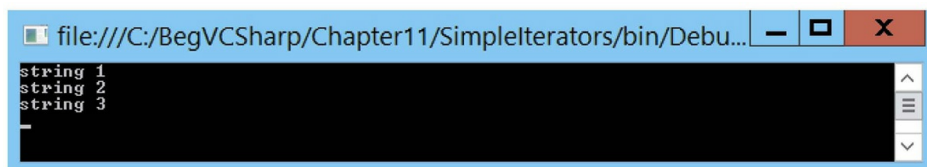


图11-3

显然，这个迭代器并不是特别有用，但它确实能够演示迭代器的机制，说明实现迭代器有多么简单。看看代码，读者可能会疑惑代码是如何知道返回string类型的项。实际上，并没有返回string类型的项，而是返回了object类型的值。因为object是所有类型的基类，所以可从yield语句中返回任意类型。

但编译器的智能程度很高，所以我们可以把返回值解释为foreach循环需要的任何类型。这里代码需要string类型的值，所以这就是我们要使用的值。如果修改一行yield代码，让它返回一个整数，就会在foreach循环中出现一个类型转换异常。

对于迭代器，还有一点要注意。可以使用下面的语句中断将信息返回给foreach循环的过程：

```
yield break;
```

在遇到迭代器中的这个语句时，迭代器的处理会立即中断，使用该迭代器的foreach循环也一样。

下面是一个较复杂但很有用的示例。在这个示例中，要实现一个迭代器，获取素数。

试一试：实现一个迭代器： **Ch11Ex03**

（1）在C:\BegVCSharp\Chapter11目录中创建一个新控制台应用程序Ch11Ex03。

（2）添加一个新类Primes，修改Primes.cs中的代码，如下所示：

```
using System;
using System.Collections;

using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Ch11Ex03
{
    public

class Primes
{
```



```
private long min;
```

```
private long max;
```

```
public Primes() : this(2, 100) {}
```

```
public Primes(long minimum, long maximum)
```

```
{
```

```
    if (minimum<2)
```

```
        min = 2;
```

```
else
```

```
    min = minimum;
```

```
    max = maximum;
```

```
}
```

```
public IEnumerator GetEnumerator()
```

```
{
```

```
    for (long possiblePrime = min; possiblePrime<= max; pos
```

```
{
```

```
    bool isPrime = true;
```

```
    for (long possibleFactor = 2; possibleFactor <=
```

```
        (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)
```

```
    {
```

```
        long remainderAfterDivision = possiblePrime % possibleFactor;
```

```
        if (remainderAfterDivision == 0)
```

```
{
```

```
    isPrime = false;
```

```
    break;
```

```
}
```

```
}
```

```
if (isPrime)
```

```
        {  
  
            yield return possiblePrime;  
  
        }  
  
    }  
  
    }  
  
    }  
  
    }  
}
```

(3) 修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)  
{
```

```
Primes primesFrom2To1000 = new Primes(2, 1000);
```

```
foreach (long i in primesFrom2To1000)
```

```
Write($"{i} ");
```

```
ReadKey();
```

```
}
```

(4) 执行应用程序，结果如图11-4所示。

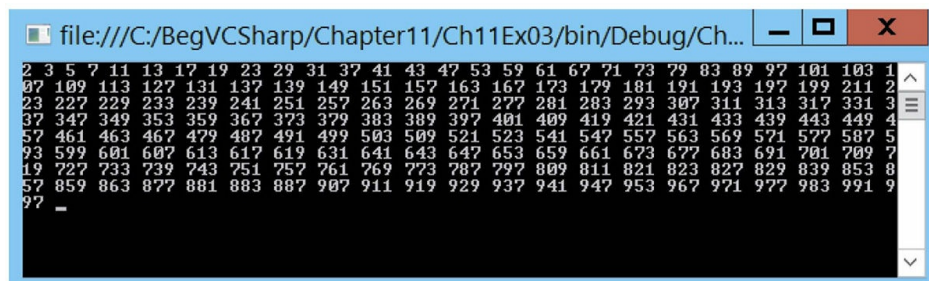


图11-4

示例的说明

这个示例中的类可以枚举上下限之间的素数集合。封装素数的类利用迭代器提供了这个功能。

Primes的代码开始时比较简单，用两个字段来存储表示搜索范围的最大值和最小值，并使用构造函数设置这些值。注意，最小值是有限制的，它不能小于2，这很合理，因为2是最小的素数。相关的代码则全部放在方法GetEnumerator()中。该方法的签名满足迭代器块的规则，因为它返回IEnumerator类型：

```
public IEnumerator GetEnumerator()  
{
```

为提取上下限之间的素数，需要依次测试每个值，所以用一个for循环开始：

```
    for (long possiblePrime = min; possiblePrime <= max; possiblePrime++)  
    {
```

由于我们不知道某个数是不是素数，所以先假定这个数是素数，再看看它是否不是素数。为此，需要看看该数能否被2到该数平方根之间的所有数整除。如果能，则该数不是素数，于是测试下一个数。如果该数的确是素数，就使用yield把它传送给foreach循环。

```
        bool isPrime = true;  
        for (long possibleFactor = 2; possibleFactor <= (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)  
        {
```

```
        long remainderAfterDivision = possiblePrime % possibleDivisor;
        if (remainderAfterDivision == 0)
        {
            isPrime = false;
            break;
        }
    }
    if (isPrime)
    {
        yield return possiblePrime;
    }
}
```

这段代码有一个有趣之处：如果把上下限设置为非常大的数，在执行应用程序时，就会发现，会一次显示一个结果，中间有暂停，而不是一次显示所有结果。这说明，无论代码在yield调用之间是否终止，迭代器代码都会一次返回一个结果。在后台，调用yield都会中断代码的执行，当请求另一个值时，也就是当使用迭代器的foreach循环开始一个新循环时，代码会恢复执行。

11.1.7 迭代器和集合

前面曾提到，将介绍迭代器如何用于迭代存储在字典类型的集合中的对象，而不必处理DictionaryItem对象。在本章下载代码的

DictionaryAnimals文件夹中，可以找到接下来这个项目的代码。下面是集合类Animals:

```
public class Animals : DictionaryBase
{
    public void Add(string newID, Animal newAnimal)
    {
        Dictionary.Add(newID, newAnimal);
    }
    public void Remove(string animalID)
    {
        Dictionary.Remove(animalID);
    }
    public Animal this[string animalID]
    {
        get { return (Animal)Dictionary[animalID]; }
        set { Dictionary[animalID] = value; }
    }
}
```

可在这段代码中添加如下的简单迭代器，以便执行预期的操作:

```
public new IEnumerator GetEnumerator()
{
    foreach (object animal in Dictionary.Values)
        yield return (Animal)animal;
}
```

现在可使用下面的代码来迭代集合中的Animal对象了：

```
foreach (Animal myAnimal in animalCollection)
{
    WriteLine($"New {myAnimal.ToString()} object added to "
        $" custom collection, Name = {myAnimal.Name}");
}
```

11.1.8 深度复制

第9章通过下面的GetCopy()方法，介绍了如何使用受保护的方法System.Object.MemberwiseClone()进行浅度复制。

```
public class Cloner
{
    public int Val;
    public Cloner(int newVal)
    {
        Val = newVal;
    }
    public object GetCopy() => MemberwiseClone();
}
```

假定有引用类型的字段，而不是值类型的字段（例如，对象）：

```
public class Content
```

```
{
```

```
    public int Val;
```

```
}
```

```
public class Cloner
```

```
{
```

```
    public Content MyContent = new Content();
```

```
    public Cloner(int newVal)
```

```
    {
```

```
        MyContent.Val = newVal;
```

```
    }
```

```
    public object GetCopy() => MemberwiseClone();
```

```
}
```

此时，通过GetCopy()得到的浅度复制包括一个字段，它引用的对象与源对象相同。以下代码使用这个Cloner类来说明浅度复制引用类型的结果：

```
Cloner mySource = new Cloner(5);
Cloner myTarget = (Cloner)mySource.GetCopy();
WriteLine($"myTarget.MyContent.Val = {myTarget.MyContent.Val}"
mySource.MyContent.Val = 2;
WriteLine($"myTarget.MyContent.Val = {myTarget.MyContent.Val}"
```

第4行把一个值赋给mySource.MyContent.Val，它是源对象中公共字段MyContent的公共字段Val。这也改变了myTarget.MyContent.Val的值。这是因为mySource.MyContent引用了与myTarget.MyContent相同的对象实例。上述代码的输出结果如下：

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 2
```

为解决这个问题，需要执行深度复制。修改上面的GetCopy()方法就可以进行深度复制，但最好使用.NET Framework的标准方式：实现ICloneable接口，该接口有一个方法Clone()；这个方法不带参数，返回一个object类型的结果，其签名和上面使用的GetCopy()方法相同。

为修改上面的类，可使用下面的深度复制代码：

```
public class Content
{
    public int Val;
}
```

```
public class Cloner : ICloneable
```

```
{  
    public Content MyContent = new Content();  
    public Cloner(int newVal)  
    {  
        MyContent.Val = newVal;  
    }  
    public object Clone()
```

```
{
```

```
    Cloner clonedCloner = new Cloner(MyContent.Val);
```

```
    return clonedCloner;
```

```
}
```

```
}
```

其中使用包含在源Cloner对象中的Content对象（MyContent）的Val字段，创建一个新Cloner对象。这个字段是一个值类型，所以不需要深度复制。

使用与上面类似的代码来测试浅度复制，但用Clone()替代GetCopy()，得到如下结果：

```
myTarget.MyContent.Val = 5  
myTarget.MyContent.Val = 5
```

这次包含的对象是独立的。注意有时在比较复杂的对象系统中，调用Clone()是一个递归过程。例如，如果Cloner类的MyContent字段也需要深度复制，就要使用下面的代码：

```
public class Cloner : ICloneable  
{  
    public Content MyContent = new Content();  
    ...  
    public object Clone()  
  
    {
```

```
        Cloner clonedCloner = new Cloner();

        clonedCloner.MyContent = MyContent.Clone();

        return clonedCloner;

    }

}
```

这里调用了默认的构造函数，以便简化创建一个新Cloner对象的语法。为使这段代码能正常工作，还需要在Content类上实现ICloneable接口。

11.1.9 给CardLib添加深度复制

下面把上述内容付诸于实践：使用ICloneable接口，复制Card、Cards和Deck对象，这在某些扑克牌游戏中是有用的，因为在这些游戏中不需要让两副扑克牌引用同一组Card对象，但肯定会使一副扑克牌中的牌序与另一副牌的牌序相同。

在Ch11CardLib中，对Card类执行复制操作是很简单的，因为只需要进行浅度复制（Card只包含值类型的数据，其形式为字段）。我们只需要对类定义进行如下修改：

```
public class Card : ICloneable

{

    public object Clone() => MemberwiseClone();
}
```

ICloneable接口的这段实现代码只是一个浅度复制，无法确定在Clone()方法中执行了什么操作，所以这已经足以满足我们的需要。

接着，需要对Cards集合类实现ICloneable接口。这个过程稍复杂些，因为涉及到复制源集合中的每个Card对象，所以需要进行深度复制：

```
public class Cards : CollectionBase, ICloneable
```



```
{
```

```
    public object Clone()
```

```
{
```

```
    Cards newCards = new Cards();
```

```
    foreach (Card sourceCard in List)
```

```
{
```

```
        newCards.Add((Card)sourceCard.Clone());
```

```
}
```

```
return newCards;
```

```
}
```

最后，需要在Deck类上实现ICloneable接口。这里存在一个小问题：因为Ch11CardLib中的Deck类无法修改它包含的扑克牌，所以没有洗牌。例如，无法修改有给定牌序的Deck实例。为解决这个问题，为Deck类定义一个新的私有构造函数，在实例化Deck对象时，可以给该函数传送指定的Cards集合。所以，在这个类中执行复制的代码如下所示：

```
public class Deck : ICloneable
```

```
{
```

```
public object Clone()
```

```
{
```

```
    Deck newDeck = new Deck(cards.Clone() as Cards);
```

```
    return newDeck;
```

```
}
```

```
private Deck(Cards newCards)
```

```
{
```

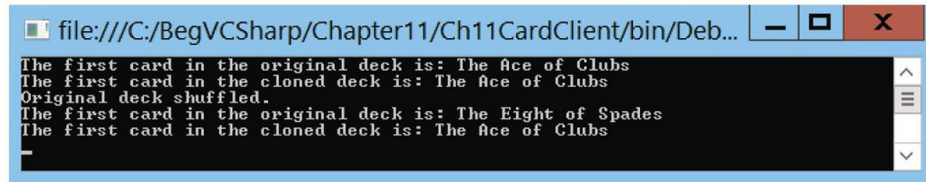
```
        cards = newCards;

    }
}
```

再次用一些简单的客户代码进行测试。与以前一样，这应放在客户项目的Main()方法中，以便进行测试（包含在本章下载代码的Ch11CardClient\Program.cs文件中）：

```
Deck deck1 = new Deck();
Deck deck2 = (Deck)deck1.Clone();
WriteLine($"The first card in the original deck is: {deck1.Get
WriteLine($"The first card in the cloned deck is: {deck2.GetCa
deck1.Shuffle();
WriteLine("Original deck shuffled.");
WriteLine($"The first card in the original deck is: {deck1.Get
WriteLine($"The first card in the cloned deck is: {deck2.GetCa
ReadKey();
```

其输出结果如图11-5所示。



```
file:///C:/BegVCSharp/Chapter11/Ch11CardClient/bin/Deb...
The first card in the original deck is: The Ace of Clubs
The first card in the cloned deck is: The Ace of Clubs
Original deck shuffled.
The first card in the original deck is: The Eight of Spades
The first card in the cloned deck is: The Ace of Clubs
```

图11-5

11.2 比较

本节介绍对象之间的两类比较：

- 类型比较
- 值比较

类型比较确定对象是什么，或者对象继承了什么，在C#编程中，这是非常重要的。把对象传送给方法时，下一步要执行什么操作常取决于对象的类型。本章和前面的章节都讨论过传送对象的内容，这里将介绍一些更有用的技巧。

“值比较”我们也见过许多，至少见过简单类型的值比较。在比较对象的值时，情况会变得较为复杂：必须从一开始就定义比较的含义，确定像>这样的运算符在比较类时会执行什么操作。这在集合中尤其重要，有时我们希望根据某个条件排列对象的顺序，例如按照字母顺序或者根据某个比较复杂的算法来排序。

11.2.1 类型比较

在比较对象时，常需要了解它们的类型，才能确定是否可以进行值的比较。第9章介绍了GetType()方法，所有的类都从System.Object中继承了这个方法，这个方法和typeof()运算符一起使用，就可以确定对象的类型（并据此执行操作）：

```
if (myObj.GetType() == typeof(MyComplexClass))
{
    // myObj is an instance of the class MyComplexClass.
}
```

前面还提到ToString()的默认实现方式，ToString()也是从System.Object继承来的，该方法可以提供对象类型的字符串表示。也可以比较这些字符串，但这是比较杂乱的方式。

本节将介绍比较值的一种简便方式：is运算符。它可以提供可读性较高的代码，还可以检查基类。在介绍is运算符之前，需要了解处理值类型（与引用类型相反）时后台的一些常见操作：封箱（boxing）和拆箱（unboxing）。

1. 封箱和拆箱

第8章讨论了引用类型和值类型之间的区别，第9章通过比较结构（值类型）和类（引用类型）展示了这些区别。封箱（boxing）是把值类型转换为System.Object类型，或者转换为由值类型实现的接口类型。拆箱（unboxing）是相反的过程。

例如，下面的结构类型：

```
struct MyStruct
{
    public int Val;
}
```

可以把这种类型的结构放在object类型的变量中，对其封箱：

```
MyStruct valType1 = new MyStruct();  
valType1.Val = 5;  
object refType = valType1;
```

其中创建了一个类型为MyStruct的新变量valType1，并把一个值赋予这个结构的Val成员，然后把它封箱在object类型的变量refType中。

以这种方式封箱变量而创建的对象，会包含值类型变量的一个副本的引用，而不包含源值类型变量的引用。要进行验证，可以修改源结构的内容，把对象中包含的结构拆箱到新变量中，检查其内容：

```
valType1.Val = 6;  
MyStruct valType2 = (MyStruct)refType;  
WriteLine($"valType2.Val = {valType2.Val}");
```

执行这段代码将得到如下输出结果：

```
valType2.Val = 5
```

但在把一个引用类型赋予对象时，将执行不同的操作。把MyStruct改为一个类（不考虑这个类名不合适的情况），即可看到这种情形：

class

```
MyStruct  
{  
    public int Val;
```



```
}
```

如果不修改上面的客户代码（再次忽略名称有误的变量），就会得到如下输出结果：

```
valType2.Val = 6
```

也可以把值类型封箱到接口类型中，只要它们实现这个接口即可。例如，假定MyStruct类型实现IMyInterface接口，如下所示：

```
interface IMyInterface {}
```

```
struct MyStruct : IMyInterface
```

```
{  
    public int Val;  
}
```

接着把结构封箱到一个IMyInterface类型中，如下所示：

```
MyStruct valType1 = new MyStruct();  
IMyInterface refType = valType1;
```

然后使用一般的数据类型转换语法对其拆箱：

```
MyStruct ValType2 = (MyStruct)refType;
```

从这些示例中可以看出，封箱是在没有用户干涉的情况下进行的（即不需要编写任何代码），但拆箱一个值需要进行显式转换，即需要进行数据类型转换（封箱是隐式的，所以不需要进行数据类型转换）。

读者可能想知道为什么要这么做。封箱非常有用，有两个非常重要的原因。首先，它允许在项的类型是object的集合（如ArrayList）中使用值类型。其次，有一个内部机制允许在值类型（例如int和结构）上调用object方法。

最后需要注意的是，在访问值类型内容前，必须进行拆箱。

2. is运算符

is运算符并不是用来说明对象是某种类型，而是用来检查对象是不是给定类型，或者是否可以转换为给定类型，如果是，这个运算符就返回true。

在前面的示例中，有Cow和Chicken类，它们都继承于Animal。使用is运算符比较Animal类型的对象，如果对象是这3种类型中的一种（不仅是Animal），is运算符就返回true。使用前面介绍的GetType()方法和typeof()运算符很难做到这一点。

is运算符的语法如下：

<operand

> `is<type`

>

这个表达式的结果如下：

- 如果<*type*>是一个类类型，而<*operand*>也是该类型，或者它继承了该类型，或者它可以封箱到该类型中，则结果为true。
- 如果<*type*>是一个接口类型，而<*operand*>也是该类型，或者它是实现该接口的类型，则结果为true。
- 如果<*type*>是一个值类型，而<*operand*>也是该类型，或者它可以拆箱到该类型中，则结果为true。

下面用一个示例说明如何使用该运算符。

试一试：使用**is**运算符：**Ch11Ex04\Program.cs**

（1）在C:\BegVCSharp\Chapter11目录中创建一个新控制台应用程序Ch11Ex04。

（2）修改Program.cs中的代码，如下所示：

```
namespace Ch11Ex04
{
    class Checker
```

```
{
```

```
public void Check(object param1)
```

```
{
```

```
if (param1 is ClassA)
```

```
WriteLine("Variable can be converted to ClassA.");
```

```
else
```

```
WriteLine("Variable can't be converted to ClassA.");
```

```
if (param1 is IMyInterface)
```

```
    WriteLine("Variable can be converted to IMyInterface.")
```

```
else
```

```
    WriteLine("Variable can't be converted to IMyInterface.
```

```
if (param1 is MyStruct)
```

```
    WriteLine("Variable can be converted to MyStruct.");
```

else

WriteLine("Variable can't be converted to MyStruct.");

}

}

interface IMyInterface {}

class ClassA : IMyInterface {}

class ClassB : IMyInterface {}

```
class ClassC {}
```

```
class ClassD : ClassA {}
```

```
struct MyStruct : IMyInterface {}
```

```
class Program  
{  
    static void Main(string[] args)  
    {  
        Checker check = new Checker();
```

```
ClassA try1 = new ClassA();
```

```
ClassB try2 = new ClassB();
```

```
ClassC try3 = new ClassC();
```

```
ClassD try4 = new ClassD();
```

```
MyStruct try5 = new MyStruct();
```

```
object try6 = try5;
```

```
WriteLine("Analyzing ClassA type variable:");
```

```
check.Check(try1);
```

```
WriteLine("\nAnalyzing ClassB type variable:");
```



```
check.Check(try2);
```

```
WriteLine("\nAnalyzing ClassC type variable:");
```

```
check.Check(try3);
```

```
WriteLine("\nAnalyzing ClassD type variable:");
```

```
check.Check(try4);
```

```
WriteLine("\nAnalyzing MyStruct type variable:");
```

```
check.Check(try5);
```

```
WriteLine("\nAnalyzing boxed MyStruct type variable:");
```

```
check.Check(try6);
```

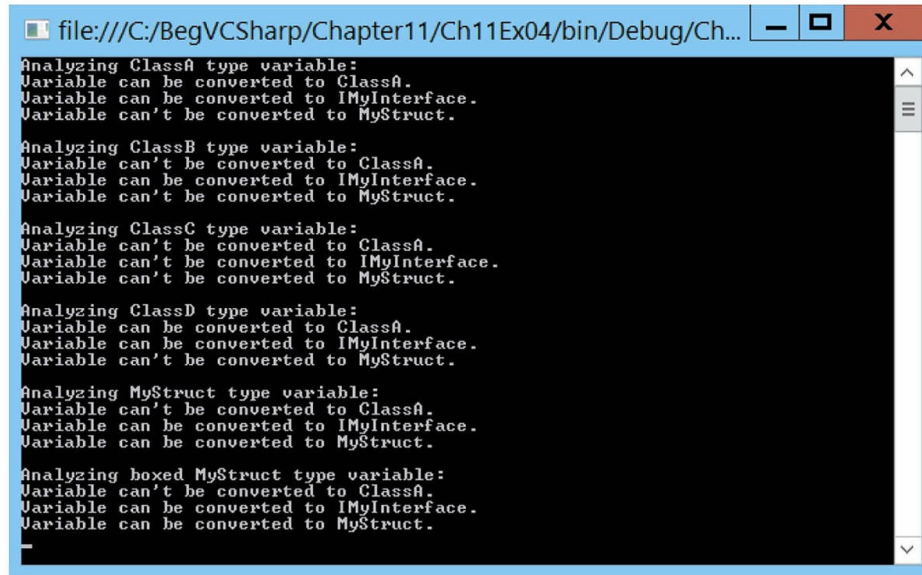
```
ReadKey();
```

```
}
```

```
}
```

```
}
```

(3) 运行代码，其结果如图11-6所示。



```
file:///C:/BegVCSharp/Chapter11/Ch11Ex04/bin/Debug/Ch...
Analyzing ClassA type variable:
Variable can be converted to ClassA.
Variable can be converted to IMyInterface.
Variable can't be converted to MyStruct.

Analyzing ClassB type variable:
Variable can't be converted to ClassA.
Variable can be converted to IMyInterface.
Variable can't be converted to MyStruct.

Analyzing ClassC type variable:
Variable can't be converted to ClassA.
Variable can't be converted to IMyInterface.
Variable can't be converted to MyStruct.

Analyzing ClassD type variable:
Variable can be converted to ClassA.
Variable can be converted to IMyInterface.
Variable can't be converted to MyStruct.

Analyzing MyStruct type variable:
Variable can't be converted to ClassA.
Variable can be converted to IMyInterface.
Variable can be converted to MyStruct.

Analyzing boxed MyStruct type variable:
Variable can't be converted to ClassA.
Variable can be converted to IMyInterface.
Variable can be converted to MyStruct.
```

图11-6

示例的说明

这个示例说明了使用is运算符的各种可能结果。其中定义了3个类、一个接口和一个结构，并把它们用作一个类的方法的参数，这个类使用is运算符确定它们是否可以转换为ClassA类型、接口类型和结构类型。

只有ClassA和ClassD（继承了ClassA）类型与ClassA兼容。如果一个类型没有继承一个类，该类型不会与该类兼容。

ClassA、ClassB和MyStruct类型都实现了IMyInterface，所以它们都与IMyInterface类型兼容。ClassD继承了ClassA，所以它们两个也兼容。因此，只有ClassC是不兼容的。

最后，只有MyStruct类型本身的变量和该类型的封箱变量与MyStruct兼容，因为不能把引用类型转换为值类型（当然，我们能够拆箱以前封箱的变量）。

11.2.2 值比较

考虑两个表示人的`Person`对象，它们都有一个`Age`整型属性。下面要比较它们，看看哪个人年龄较大。为此可以使用以下代码：

```
if (person1.Age > person2.Age)
{
    ...
}
```

这是可以的，但还有其他方法，例如，使用下面的语法：

```
if (person1 > person2)
{
    ...
}
```

可以使用运算符重载，如本节后面所述。这是一项强大的技术，但应谨慎使用。在上面的代码中，年龄的比较不是非常明显，该段代码还可以比较身高、体重、IQ等。

另一个方法是使用`Comparable`和`Comparer`接口，它们可采用标准方式定义比较对象的过程。`.NET Framework`中的各种集合类支持这种方式，这使得它们成为对集合中的对象进行排序的一种极佳方式。

1. 运算符重载

通过运算符重载（operator overloading），可以对我们设计的类使用标准的运算符，例如+、>等。这称为重载，因为在使用特定的参数类型时，我们为这些运算符提供了自己的实现代码，其方式与重载方法相同，也是为同名方法提供不同的参数。

运算符重载非常有用，因为我们可以再运算符重载的实现中执行需要的任何操作，这并不一定像用“+”表示“把这两个操作数相加”这么简单。稍后介绍一个进一步升级CardLib库的示例。我们将提供比较运算符的实现代码，比较两张牌，看看在一圈（扑克牌游戏中的一局）中哪张牌会赢。

因为在许多扑克牌游戏中，一圈取决于牌的花色，这并不像比较牌上的数字那样直接。如果第二张牌与第一张牌的花色不同，则无论其点数是什么，第一张牌都会赢。考虑两个操作数的顺序，就可以实现这种比较。也可以考虑“王牌”的花色，而王牌可以胜过其他花色，即使该王牌的花色与第一张牌不同，也是如此。也就是说，`card1>card2`是`true`（这表示如果`card1`是第一个出牌，则`card1`胜过了`card2`），并不意味着`card2>card1`是`false`。如果`card1`和`card2`都不是王牌，且属于不同的花色，则这两个比较都是`true`。

但我们先看一下运算符重载的基本语法。要重载运算符，可给类添加运算符类型成员（它们必须是`static`）。一些运算符有多种用途（如-运算符就有一元和二元两种功能），因此我们还指定了要处理多少个操作数，以及这些操作数的类型。一般情况下，操作数的类型与定义运算符的类相同，但也可以定义处理混合类型的运算符，详见稍后的内容。

例如，考虑一个简单类型AddClass1，如下所示：

```
public class AddClass1
{
    public int val;
}
```

这仅是int值的一个包装器（wrapper），但可以用于说明原理。对于这个类，下面的代码不能编译：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
AddClass1 op3 = op1 + op2;
```

其错误是+运算符不能应用于AddClass1类型的操作数，因为我们尚未定义要执行的操作。下面的代码则可执行，但无法得到预期的结果：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
bool op3 = op1 == op2;
```

其中，使用==二元运算符来比较op1和op2，看它们是否引用同一个对象，而不是验证它们的值是否相等。在上述代码中，即使op1.val和

op2.val相等，op3也是false。

要重载+运算符，可使用下述代码：

```
public class AddClass1
{
    public int val;
    public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
    {
        AddClass1 returnVal = new AddClass1();

        returnVal.val = op1.val + op2.val;

        return returnVal;
    }
}
```

```
}
```

```
}
```

可以看出，运算符重载看起来与标准静态方法声明类似，但它们使用关键字`operator`和运算符本身，而不是一个方法名。现在可以成功地使用`+`运算符和这个类，如上面的示例所示：

```
AddClass1 op3 = op1 + op2;
```

重载所有的二元运算符都是一样的，一元运算符看起来也是类似的，但只有一个参数：

```
public class AddClass1
{
    public int val;
    public static AddClass1 operator +(AddClass1 op1, AddClass1
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }
    public static AddClass1 operator -(AddClass1 op1)
```



```
{  
  
    AddClass1 returnVal = new AddClass1();  
  
    returnVal.val = -op1.val;  
  
    return returnVal;  
  
}  
  
}
```

这两个运算符处理的操作数的类型与类相同，返回值也是该类型，但考虑下面的类定义：

```
public class AddClass1  
{
```

```

    public int val;
    public static AddClass3 operator +(AddClass1 op1, AddClass2
    {
        AddClass3 returnVal = new AddClass3();
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }
}
public class AddClass2
{
    public int val;
}
    public class AddClass3
    {
        public int val;
    }
}

```

下面的代码就可以执行：

```

AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass2 op2 = new AddClass2();
op2.val = 5;
AddClass3 op3 = op1 + op2;

```

可以酌情采用这种方式混合类型。但要注意，如果把相同的运算符添加到AddClass2中，上面的代码就会失败，因为它弄不清要使用哪个运算符。因此，应注意不要把签名相同的运算符添加到多个类中。

还要注意，如果混合了类型，操作数的顺序必须与运算符重载的参数顺序相同。如果使用了重载的运算符和顺序错误的操作数，操作就会失败。所以不能像下面这样使用运算符：

```
AddClass3 op3 = op2 + op1;
```

当然，除非提供了另一个重载运算符和倒序的参数：

```
public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)
{
    AddClass3 returnVal = new AddClass3();
    returnVal.val = op1.val + op2.val;
    return returnVal;
}
```

可以重载下述运算符：

- 一元运算符： +, -, !, ~, ++, --, true, false
- 二元运算符： +, -, *, /, %, &, |, ^, <<, >>
- 比较运算符： ==, !=, <, >, <=, >=

注意： 如果重载true和false运算符，就可以在布尔表达式中使用类，例如，if (op1) {}。

不能重载赋值运算符，例如+=，但这些运算符使用与它们对应的简单运算符，例如+，所以不必担心它们。重载+意味着+=如期执行。=运算符不能重载，因为它有一个基本的用途。但这个运算符与用户定义的

转换运算符相关，详见下一节。

也不能重载&&和||，但它们使用对应的运算符&和|执行计算，所以重载&和|就足够了。

一些运算符（如<和>）必须成对重载。这就是说，如果重载>，就必须也重载<。许多情况下，可以在这些运算符中调用其他运算符，以减少需要的代码数量（和可能发生的错误），例如：

```
public class AddClass1
{
    public int val;
    public static bool operator >=(AddClass1 op1, AddClass1 op2
```

```
    => (op1.val >= op2.val);
```

```
    public static bool operator <(AddClass1 op1, AddClass1 op2)
```

```
    => !(op1 >= op2);
```

}

这同样适用于==和!=，但对于这些运算符，通常需要重写Object.Equals()和Object.GetHashCode()，因为这两个函数也可以用于比较对象。重写这些方法，可以确保无论类的用户使用什么技术，都能得到相同的结果。这不太重要，但应增加进来，以保证其完整性。它需要下述非静态重写方法：

```
public class AddClass1
{
    public int val;

    public static bool operator ==(AddClass1 op1, AddClass1 op2)
        => (op1.val == op2.val);

    public static bool operator !=(AddClass1 op1, AddClass1 op2)
        => !(op1 == op2);

    public override bool Equals(object op1) => val == ((AddClass1)op1).val;

    public override int GetHashCode() => val;
}
```

```
}
```

GetHashCode()可根据其状态，获取对象实例的一个唯一int值。这里使用val就可以了，因为它也是一个int值。

注意，Equals()使用object类型参数。我们需要使用这个签名，否则就将重载这个方法，而不是重写它。类的用户仍可以访问默认的实现代码。这样就必须使用数据类型转换得到所需的结果。这常需要使用本章前面讨论的is运算符检查对象类型，代码如下所示：

```
public override bool Equals(object op1)
{
    if (op1 is AddClass1)

    {

        return val == ((AddClass1)op1).val;

    }
}
```

else

{

throw new ArgumentException(

"Cannot compare AddClass1 objects with objects of type

+ op1.GetType().ToString());

}

}

在这段代码中，如果传送给Equals的操作数的类型有误，或者不能转换为正确类型，就会抛出一个异常。当然，这可能并不是我们希望的操作。我们要比较一个类型的对象和另一个类型的对象，此时需要更多的分支结构。另外，可能只允许对类型完全相同的两个对象进行比较，这需要对第一个if语句做如下修改：

```
if (op1.GetType() == typeof(AddClass1))
```

2. 给CardLib添加运算符重载

现在再次升级Ch11CardLib项目，给Card类添加运算符重载。在本章下载代码的Ch11CardLib文件夹中可以找到以下的类的代码。首先给Card类添加额外字段，指定某花色比其他花色大，使A有更高的级别。把这些字段指定为静态，因为设置它们后，它们就可以应用到所有Card对象上：

```
public class Card
{
    ///<summary>
```

```
    /// Flag for trump usage. If true, trumps are valued highe
```



```
/// than cards of other suits.
```

```
///</summary>
```

```
public static bool useTrumps = false;
```

```
///<summary>
```

```
/// Trump suit to use if useTrumps is true.
```

```
///</summary>
```

```
public static Suit trump = Suit.Club;
```

```
///<summary>
```

```
/// Flag that determines whether aces are higher than king
```

```
/// than deuces.
```

```
///</summary>
```

```
public static bool isAceHigh = true;
```

这些规则应用于应用程序中每个Deck的所有Card对象上。因此，两个Deck中的Card不可能遵守不同规则。这适用于这个类库，但是确实可以做出这样的假设：如果一个应用程序要使用不同的规则，可以自行维护这些规则；例如，在切换牌时，设置Card的静态成员。

完成后，就要给Deck类再添加几个构造函数，以便用不同的特性来初始化扑克牌：

```
///<summary>
```

```
/// Nondefault constructor. Allows aces to be set high.
```

```
///</summary>
```

```
public Deck(bool isAceHigh) : this()
```

```
{
```

```
    Card.isAceHigh = isAceHigh;
```

```
}
```

```
///<summary>
```

```
/// Nondefault constructor. Allows a trump suit to be u
```

```
///</summary>
```

```
public Deck(bool useTrumps, Suit trump) : this()
```

```
{
```

```
    Card.useTrumps = useTrumps;
```

```
Card.trump = trump;
```

```
}
```

```
///<summary>
```

```
/// Nondefault constructor. Allows aces to be set high
```

```
/// to be used.
```

```
///</summary>
```

```
public Deck(bool isAceHigh, bool useTrumps, Suit trump)
```

```
{  
  
    Card.isAceHigh = isAceHigh;  
  
    Card.useTrumps = useTrumps;  
  
    Card.trump = trump;  
  
}
```

每个构造函数都使用第9章介绍的: `this()`语法来定义，这样，无论如何，默认构造函数总会在非默认的构造函数之前调用，初始化扑克牌。

注意： 第12章将详细讨论==和>操作符重载方法实现的空条件操作符（?.）。在这个代码段中，`public static bool operator==`方法的`card1?.suit`在试图检索suit存储的值之前，检查card1对象是否为空。在以后的章节中实现方法时，这是很重要的。

接着，给Card类添加运算符重载（和推荐的重写代码）：

```
public static bool operator ==(Card card1, Card card2)
```

```
=> card1?.suit == card2?.suit) && (card1?.rank == card2?.r
```

```
public static bool operator !=(Card card1, Card card2)
```

```
=> !(card1 == card2);
```

```
public override bool Equals(object card) => this == (Card)ca
```

```
public override int GetHashCode()
```

```
    => return 13 * (int)suit + (int)rank;
```

```
public static bool operator >(Card card1, Card card2)
```

```
{
```

```
    if (card1.suit == card2.suit)
```

```
{
```

```
    if (isAceHigh)
```



```
{
```

```
    if (card1.rank == Rank.Ace)
```

```
    {
```

```
        if (card2.rank == Rank.Ace)
```

```
            return false;
```

```
    else
```

```
return true;
```

```
}
```

```
else
```

```
{
```

```
if (card2.rank == Rank.Ace)
```

```
return false;
```

```
else
```

```
return (card1.rank > card2?.rank);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
return (card1.rank > card2.rank);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
    if (useTrumps && (card2.suit == Card.trump))
```

```
        return false;
```

```
    else
```

```
        return true;
```

```
}
```

```
}
```

```
public static bool operator<(Card card1, Card card2)
```

```
=> !(card1 >= card2);
```

```
public static bool operator >=(Card card1, Card card2)
```

```
{
```

```
if (card1.suit == card2.suit)
```

```
{
```

```
    if (isAceHigh)
```

```
    {
```

```
        if (card1.rank == Rank.Ace)
```

```
        {
```

```
            return true;
```

```
}
```

```
else
```

```
{
```

```
    if (card2.rank == Rank.Ace)
```

```
        return false;
```

```
else
```

```
    return (card1.rank >= card2.rank);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
    return (card1.rank >= card2.rank);
```

```
}
```

```
}
```


else

{

if (useTrumps && (card2.suit == Card.trump))

return false;

else

return true;

```
}
```

```
}
```

```
public static bool operator <=(Card card1, Card card2)
```

```
=> !(card1 > card2);
```

这段代码没什么需要特别关注之处，只是>和>=重载运算符的代码比较长。如果单步执行>运算符的代码，就可以看到它的执行情况，明白为什么需要这些步骤。

比较两张牌card1和card2，其中card1假定为先出的牌。如前所述，在使用王牌时，这是很重要的，因为王牌胜过其他牌，即使非王牌比较大，也是这样。当然，如果两张牌的花色相同，则王牌是否也是该花色就不重要了，所以这是我们要进行的第一个比较：

```
public static bool operator >(Card card1, Card card2)
```

```
{
```

```
if (card1.suit == card2.suit)
{
```

如果静态的isAceHigh标记为true，就不能直接通过Rank枚举中的值比较牌的点数了。因为A的点数在这个枚举中是1，比其他牌都小。此时就需要如下步骤：

- 如果第一张牌是A，就检查第二张牌是否也是A。如果是，则第一张牌就胜不过第二张牌。如果第二张牌不是A，则第一张牌胜出：

```
if (isAceHigh)
{
    if (card1.rank == Rank.Ace)
    {
        if (card2.rank == Rank.Ace)
            return false;
        else
            return true;
    }
}
```

- 如果第一张牌不是A，也需要检查第二张牌是不是A。如果是，则第二张牌胜出；否则，就可以比较牌的点数，因为此时已不比较A了：

```
else
{
    if (card2.rank == Rank.Ace)
        return false;
```

```

        else
            return (card1.rank > card2?.rank);
    }
}

```

- 另外，如果A不是最大的，就只需比较牌的点数：

```

    else
    {
        return (card1.rank > card2.rank);
    }

```

代码的其余部分主要考虑card1和card2花色不同的情况。其中静态useTrumps标记是非常重要的。如果这个标记是true，且card2是王牌，则可以肯定，card1不是王牌（因为这两张牌有不同的花色），王牌总是胜出，所以card2比较大：

```

    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;

```

如果card2不是王牌（或者useTrumps是false），则card1胜出，因为它是最先出的牌：

```

        else
            return true;
    }
}

```

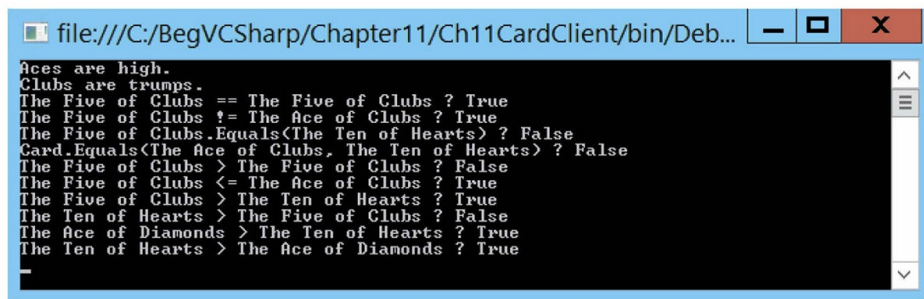
另有一个运算符（>=）使用与此类似的代码，除此之外的其他运算符都非常简单，所以不需要详细分析它们。

下面的简单客户代码测试这些运算符。把它放在客户项目的Main()方法中进行测试，就像前面CardLib示例的客户代码那样（这段代码包含在Ch11CardClient\Program.cs文件中）：

```
Card.isAceHigh = true;
WriteLine("Aces are high.");
Card.useTrumps = true;
Card.trump = Suit.Club;
WriteLine("Clubs are trumps.");
Card card1, card2, card3, card4, card5;
card1 = new Card(Suit.Club, Rank.Five);
card2 = new Card(Suit.Club, Rank.Five);
card3 = new Card(Suit.Club, Rank.Ace);
card4 = new Card(Suit.Heart, Rank.Ten);
card5 = new Card(Suit.Diamond, Rank.Ace);
WriteLine($"{card1.ToString()} == {card2.ToString()} ? {card1}");
WriteLine($"{card1.ToString()} != {card3.ToString()} ? {card1}");
WriteLine($"{card1.ToString().Equals({card4.ToString()})} ? "
    $" { card1.Equals(card4)}");
WriteLine($"Card.Equals({card3.ToString()}, {card4.ToString()})
    $" { Card.Equals(card3, card4)}");
WriteLine($"{card1.ToString()} > {card2.ToString()} ? {card1} >");
WriteLine($"{card1.ToString()}<= {card3.ToString()} ? {card1}<=");
WriteLine($"{card1.ToString()} > {card4.ToString()} ? {card1} >");
```

```
WriteLine($"{card4.ToString()} > {card1.ToString()} ? {card4 >
WriteLine($"{card5.ToString()} > {card4.ToString()} ? {card5 >
WriteLine($"{card4.ToString()} > {card5.ToString()} ? {card4 >
ReadKey();
```

其结果如图11-7所示。



```
file:///C:/BegVCSharp/Chapter11/Ch11CardClient/bin/Deb...
Aces are high.
Clubs are trumps.
The Five of Clubs == The Five of Clubs ? True
The Five of Clubs != The Ace of Clubs ? True
The Five of Clubs.Equals(The Ten of Hearts) ? False
Card.Equals(The Ace of Clubs, The Ten of Hearts) ? False
The Five of Clubs > The Five of Clubs ? False
The Five of Clubs <= The Ace of Clubs ? True
The Five of Clubs > The Ten of Hearts ? True
The Ten of Hearts > The Five of Clubs ? False
The Ace of Diamonds > The Ten of Hearts ? True
The Ten of Hearts > The Ace of Diamonds ? True
```

图11-7

这两种情况下，在应用运算符时都考虑了指定的规则。这在输出结果的最后4行中尤其明显，说明王牌总是胜过其他牌。

3. IComparable和IComparer接口

IComparable和IComparer接口是.NET Framework中比较对象的标准方式。这两个接口之间的差别如下：

- IComparable在要比较的对象的类中实现，可以比较该对象和另一个对象。
- IComparer在一个单独的类中实现，可以比较任意两个对象。

一般使用IComparable给出类的默认比较代码，使用其他类给出非

默认的比较代码。

IComparable提供了一个方法CompareTo(), 这个方法接受一个对象。例如, 在实现该方法时, 使其可以接受一个Person对象, 以便确定这个人比当前的人更年老还是更年轻。实际上, 这个方法返回一个int, 所以也可以确定第二个人与当前的人的年龄差:

```
if (person1.CompareTo(person2) == 0)
{
    WriteLine("Same age");
}
else if (person1.CompareTo(person2) > 0)
{
    WriteLine("person 1 is Older");
}
else
{
    WriteLine("person1 is Younger");
}
```

IComparer也提供一个方法Compare()。这个方法接受两个对象, 返回一个整型结果, 这与CompareTo()相同。对于支持IComparer的对象, 可使用下面的代码:

```
if (personComparer.Compare(person1, person2) == 0)
{
    WriteLine("Same age");
}
```

```

else if (personComparer.Compare(person1, person2) > 0)
{
    WriteLine("person 1 is Older");
}
else
{
    WriteLine("person1 is Younger");
}

```

这两种情况下，提供给方法的参数是System.Object类型。这意味着可以比较一个对象与其他任意类型的另一个对象。所以，在返回结果之前，通常需要进行某种类型比较，如果使用了错误类型，还会抛出异常。

.NET Framework在类Comparer上提供了IComparer接口的默认实现方式，类Comparer位于System.Collections名称空间中，可以对简单类型以及支持IComparable接口的任意类型进行特定文化的比较。例如，可通过下面的代码使用它：

```

string firstString = "First String";
string secondString = "Second String";
WriteLine($"Comparing '{firstString}' and '{secondString}', "
    $"result: {Comparer.Default.Compare(firstString, secondString)}");
int firstNumber = 35;
int secondNumber = 23;
WriteLine($"Comparing '{firstNumber}' and '{secondNumber}', "
    $"result: {Comparer.Default.Compare(firstNumber, secondNumber)}");

```


这里使用`Comparer.Default`静态成员获取`Comparer`类的一个实例，接着使用`Compare()`方法比较前两个字符串，之后比较两个整数，结果如下：

```
Comparing 'First String' and 'Second String', result: -1  
Comparing '35' and '23', result: 1
```

在字母表中，F在S的前面，所以F“小于”S，第一个比较的结果就是-1。同样，35大于23，所以结果是1。注意这里的结果并未给出相差的幅度。

在使用`Comparer`时，必须使用可以比较的类型。例如，试图比较`firstString`和`firstNumber`就会生成一个异常。

下面列出有关这个类的一些注意事项：

- 检查传送给`Comparer.Compare()`的对象，看看它们是否支持`Comparable`。如果支持，就使用该实现代码。
- 允许使用`null`值，它表示“小于”其他任意对象。
- 字符串根据当前文化来处理。要根据不同的文化（或语言）处理字符串，`Comparer`类必须使用其构造函数进行实例化，以便传送用于指定所使用的文化的`System.Globalization.CultureInfo`对象。
- 字符串在处理时要区分大小写。如果要以不区分大小写的方式来处理它们，就需要使用`CaseInsensitiveComparer`类，该类以相同的方式工作。

4. 对集合排序

许多集合类可以用对象的默认比较方式进行排序，或者用定制方法来排序。ArrayList就是一个示例，它包含方法Sort()，这个方法使用时可以不带参数，此时使用默认的比较方式，也可以给它传送IComparer接口，以比较对象对。

给ArrayList填充了简单类型时，例如整数或字符串，就会进行默认的比较。对于自己的类，必须在类定义中实现Comparable，或创建一个支持IComparer的类，来进行比较。

注意，System.Collections名称空间中的一些类（包括CollectionBase）都没有提供排序方法。如果要对派生于这个类的集合排序，就必须多做一些工作，自己给内部的List集合排序。

下面的示例说明如何使用默认的和非默认的比较方式给列表排序。

试一试：给列表排序：Ch11Ex05

（1）在C:\BegVCSharp\Chapter11目录中创建一个新控制台应用程序Ch11Ex05。

（2）添加一个新类Person，修改Person.cs中的代码，如下所示：

```
namespace Ch11Ex05
{
    public
```

```
class Person : Comparable
```

```
{
```

```
    public string Name;
```

```
    public int Age;
```

```
    public Person(string name, int age)
```

```
{
```

```
        Name = name;
```

```
Age = age;
```

```
}
```

```
public int CompareTo(object obj)
```

```
{
```

```
    if (obj is Person)
```

```
    {
```

```
        Person otherPerson = obj as Person;
```

```
return this.Age - otherPerson.Age;
```

```
}
```

```
else
```

```
{
```

```
throw new ArgumentException(
```

```
"Object to compare to is not a Person object.");
```

```
}
```

```
}
```

```
}
```

```
}
```

(3) 添加一个新类 `PersonComparerName`，修改代码，如下所示：

```
using System;
```

```
using System.Collections;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

```
using System.Threading.Tasks;
```

```
namespace Ch11Ex05
```

```
{
```

```
    public
```

```
class PersonComparerName : IComparer
```

```
{
```

```
    public static IComparer Default = new PersonComparerName()
```

```
    public int Compare(object x, object y)
```

```
    {
```

```
        if (x is Person && y is Person)
```

```
        {
```

```
return Comparer.Default.Compare(  
  
    ((Person)x).Name, ((Person)y).Name);  
  
}  
  
else  
  
{  
  
    throw new ArgumentException(  
  
        "One or both objects to compare are not Person obj
```



```
}
```

```
}
```

```
}
```

```
}
```

（4）修改Program.cs中的代码，如下所示：

```
using System;
```

```
using System.Collections;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

```
using System.Threading.Tasks;
```

```
using static System.Console;
```

```
namespace Ch11Ex05
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList list = new ArrayList();

            list.Add(new Person("Rual", 30));

            list.Add(new Person("Donna", 25));

            list.Add(new Person("Mary", 27));

            list.Add(new Person("Ben", 44));

            WriteLine("Unsorted people:");
        }
    }
}
```

```
for (int i = 0; i<list.Count; i++)
```

```
{
```

```
    WriteLine($"{(list[i] as Person).Name } {(list[i] as
```

```
    }
```

```
WriteLine();
```

```
WriteLine(
```

```
"People sorted with default comparer (by age):");
```

```
list.Sort();
```

```
for (int i = 0; i<list.Count; i++)
```

```
{
```

```
WriteLine($"{(list[i] as Person).Name } {(list[i] as
```

```
}
```

```
WriteLine();
```

```
WriteLine(
```

```
    "People sorted with nondefault comparer (by name):");
```

```
list.Sort(PersonComparerName.Default);
```

```
for (int i = 0; i<list.Count; i++)
```

```
{
```

```
    WriteLine($"{(list[i] as Person).Name } {(list[i] as
```

```
}
```

```

        ReadKey();

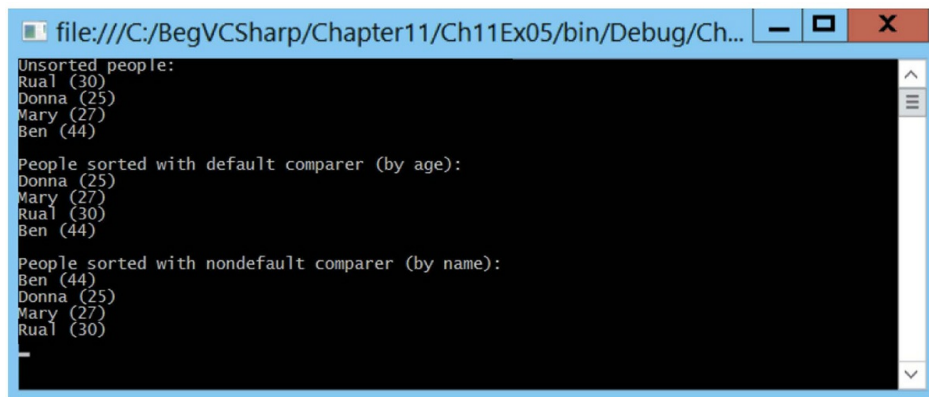
    }

}

}

```

(5) 执行代码，结果如图11-8所示。



```

file:///C:/BegVCSharp/Chapter11/Ch11Ex05/bin/Debug/Ch...
Unsorted people:
Rual (30)
Donna (25)
Mary (27)
Ben (44)

People sorted with default comparer (by age):
Donna (25)
Mary (27)
Rual (30)
Ben (44)

People sorted with nondefault comparer (by name):
Ben (44)
Donna (25)
Mary (27)
Rual (30)

```

图11-8

示例的说明

在这个示例中，包含Person对象的ArrayList用两种不同的方式排序。调用不带参数的ArrayList.Sort()方法，将使用默认的比较方式，也就是使用Person类中的CompareTo()方法（因为这个类实现了Comparable）：

```

public int CompareTo(object obj)
{
    if (obj is Person)
    {
        Person otherPerson = obj as Person;
        return this.Age - otherPerson.Age;
    }
    else
    {
        throw new ArgumentException(
            "Object to compare to is not a Person object.");
    }
}

```

这个方法首先检查其参数能否与**Person**对象比较，即该对象能否转换为**Person**对象。如果遇到问题，就抛出一个异常。否则，就比较两个**Person**对象的**Age**属性。

接着，使用实现了**IComparer**的**PersonComparerName**类，执行非默认的比较排序。这个类有一个公共的静态字段，以方便使用：

```

public static IComparer Default = new PersonComparerName()

```

它可以用**PersonComparerName.Default**获取一个实例，就像前面的**Comparer**类一样。这个类的**CompareTo()**方法如下：

```

public int Compare(object x, object y)
{

```

```
    if (x is Person && y is Person)
    {
        return Comparer.Default.Compare(
            ((Person)x).Name, ((Person)y).Name);
    }
    else
    {
        throw new ArgumentException(
            "One or both objects to compare are not Person object");
    }
}
```

这里也是首先检查参数，看看它们是不是Person对象，如果不是，就抛出一个异常；如果是，就用默认的Comparer对象比较Person对象的两个字符串字段Name。

11.3 转换

到目前为止，在需要把一种类型转换为另一种类型时，使用的都是类型转换。但这并非是唯一方式。在计算过程中，`int`可以隐式转换为`long`或`double`，采用相同的方式还可以定义所创建的类（隐式或显式）转换为其他类的方式。为此，可重载转换运算符，其方式与本章前面重载其他运算符的方式相同。本节第一部分就介绍重载方式。本节还将介绍另一个有用的运算符：`as`运算符，它一般适用于引用类型的转换。

11.3.1 重载转换运算符

除了重载上述数学运算符外，还可以定义类型之间的隐式和显式转换。如果要在不相关的类型之间转换，例如类型之间没有继承关系，也没有共享接口，就必须这么做。

下面定义`ConvClass1`和`ConvClass2`之间的隐式转换，即编写下列代码：

```
ConvClass1 op1 = new ConvClass1();  
ConvClass2 op2 = op1;
```

另外，可以定义一个显式转换：

```
ConvClass1 op1 = new ConvClass1();  
ConvClass2 op2 = (ConvClass2)op1;
```

例如，考虑下面的代码：

```
public class ConvClass1
{
    public int val;
    public static implicit operator ConvClass2(ConvClass1 op1)
    {
        ConvClass2 returnVal = new ConvClass2();
        returnVal.val = op1.val;
        return returnVal;
    }
}

public class ConvClass2
{
    public double val;
    public static explicit operator ConvClass1(ConvClass2 op1)
    {
        ConvClass1 returnVal = new ConvClass1();
        checked {returnVal.val = (int)op1.val;};
        return returnVal;
    }
}
```

其中，ConvClass1包含一个int值，ConvClass2包含一个double值。int值可以隐式转换为double值，所以可在ConvClass1和ConvClass2之间定义一个隐式转换。但反过来就不行了，应把ConvClass2和ConvClass1之间的转换定义为显式转换。

在代码中，用关键字`implicit`和`explicit`来指定这些转换，如上所示。对于这些类，下面的代码就很好：

```
ConvClass1 op1 = new ConvClass1();
op1.val = 3;
ConvClass2 op2 = op1;
```

但反向转换需要进行下述显式数据类型转换：

```
ConvClass2 op1 = new ConvClass2();
op1.val = 3e15;
ConvClass1 op2 = (ConvClass1)op1;
```

如果在显式转换中使用了`checked`关键字，则上述代码将产生一个异常，因为`op1`的`val`属性值太大，不能放在`op2`的`val`属性中。

11.3.2 `as`运算符

`as`运算符使用下面的语法，把一种类型转换为指定的引用类型：

<operand

> as<type

>

这只适用于下列情况：

- `<operand>` 的类型是 `<type>`
- `<operand>` 可以隐式转换为 `<type>` 类型
- `<operand>` 可以封箱到 `<type>` 类型中

如果不能从 `<operand>` 转换为 `<type>`，则表达式的结果就是 `null`。

基类到派生类的转换可以使用显式转换来进行，但这并不总是有效的。考虑前面示例中的两个类 `ClassA` 和 `ClassD`，其中 `ClassD` 派生于 `ClassA`：

```
class ClassA : IMyInterface {}  
class ClassD : ClassA {}
```

下面的代码使用 `as` 运算符把 `obj1` 中存储的 `ClassA` 实例转换为 `ClassD` 类型：

```
ClassA obj1 = new ClassA();  
ClassD obj2 = obj1 as ClassD;
```

则 `obj2` 的结果为 `null`。

还可以使用多态性把 `ClassD` 实例存储在 `ClassA` 类型的变量中。下面的代码演示了这一点，`ClassA` 类型的变量包含 `ClassD` 类型的实例，使用 `as` 运算符把 `ClassA` 类型的变量转换为 `ClassD` 类型。

```
ClassD obj1 = new ClassD();  
ClassA obj2 = obj1;  
ClassD obj3 = obj2 as ClassD;
```

这次obj3最后包含与obj1相同的对象引用，而不是null。

因此，as运算符非常有用，因为下面使用简单类型转换的代码会抛出一个异常：

```
ClassA obj1 = new ClassA();  
ClassD obj2 = (ClassD)obj1;
```

与此代码等价的as代码会把null值赋予obj2，不会抛出异常。这表示，下面的代码（使用本章前面开发的两个类：Animal和派生于Animal的一个类Cow）在C#应用程序中是很常见的：

```
public void MilkCow(Animal myAnimal)  
{  
    Cow myCow = myAnimal as Cow;  
    if (myCow != null)  
    {  
        myCow.Milk();  
    }  
    else  
    {  
        WriteLine($"{myAnimal.Name} isn't a cow, and so can't be r  
    }  
}
```

这要比检查异常简单得多！

11.4 练习

(1) 创建一个集合类**People**，它是下述**Person**类的集合，该集合中的项可以通过一个字符串索引符来访问，该字符串索引符是人名，与**Person.Name**属性相同：

```
public class Person
{
    private string name;
    private int age;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

(2) 扩展上一题中的**Person**类，重载>、<、>=和<=运算符，比较**Person**实例的**Age**属性。

(3) 给**People**类添加**GetOldest()**方法，使用练习（2）中定义的重

载运算符，返回其Age属性值为最大的Person对象数组（1个或多个对象，因为对于这个属性而言，多个项可以有相同的值）。

（4）在People类上实现ICloneable接口，提供深度复制功能。

（5）给People类添加一个迭代器，在下面的foreach循环中获取所有成员的年龄：

```
foreach (int age in myPeople.Ages)
{
    // Display ages.
}
```

附录A给出了练习答案。

11.5 本章要点

主题	要点
定义集合	集合是可以包含其他类的实例的类。要定义集合，可从CollectionBase中派生，或者自己实现集合接口，例如IEnumerable、ICollection和IList。一般需要为集合定义一个索引器，以使用collection[index]语法来访问集合成员
字典	也可以定义键控集合，即字典，字典中的每一项都有一个关联的键。在字典中，键可以用于标识一项，而不必使用该项目的索引。定义字典时，可以实现IDictionary，或者从DictionaryBase派生类
迭代器	可以实现一个迭代器，来控制循环代码如何在循环过程中获取值。要迭代一个类，需要实现GetEnumerator()方法，其返回类型是IEnumerator。要迭代类的成员，例如方法，可使用IEnumerable返回类型。在迭代器的代码块中，使用yield关键字返回值
类型比较	可使用GetType()方法获得对象的类型，使用typeof()运算符可以获得类的类型。可以比较这些类型值。还可以使用is运算符确定对象是否与某个类类型兼容
值比较	如果希望类的实例可以用标准的C#运算符进行比较，就必须在类定义中重载这些运算符。对于其他类型的值比较，可使用实现了Comparable或Comparer接口的类。这些接口特别适用于集合的排序
as运算符	可使用as运算符把一个值转换为引用类型。如果不能进行转换，as运算符就返回null值

第12章 泛型

本章内容：

- 泛型的含义
- 如何使用.NET Framework提供的一些泛型类
- 如何定义自己的泛型
- 变体如何与泛型一起工作

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 12 Code后，可以找到与本章示例对应的单独文件。

本章首先介绍泛型（generic）的概念，先学习抽象的泛型术语，因为学习泛型的概念对高效使用它是至关重要的。

接着讨论.NET Framework中的一些泛型类型，这有助于更好地理解其功能和强大之处，以及在代码中需要使用的新语法。然后定义自己的泛型类型，包括泛型类、接口、方法和委托。还要介绍进一步定制泛型类型的其他技术：default关键字和类型约束。

最后讨论协变（**covariance**）和抗变（**contravariance**），这是C# 4引入的两种形式的变体，在使用泛型类时提供了更大的灵活性。

12.1 泛型的含义

为介绍泛型的概念，说明它们为什么这么有用，先回顾一下第11章中的集合类。基本集合可以包含在诸如ArrayList的类中，但这些集合是没有类型化的，所以需要把object项转换为集合中实际存储的对象类型。继承自System.Object的任何对象都可以存储在ArrayList中，所以要特别仔细。假定包含在集合中的某些类型可能导致抛出异常，而且代码逻辑崩溃。前面介绍的技术可以处理这个问题，包括检查对象类型所需的代码。

但是，更好的解决办法是一开始就使用强类型化的集合类。这种集合类派生于CollectionBase，并可以拥有自己的方法，来添加、删除和访问集合的成员，但它可能把集合成员限制为派生于某种基本类型，或者必须支持某个接口。这会带来一个问题。每次创建需要包含在集合中的新类时，就必须执行下列任务之一：

- 使用某个集合类，该类已经定义为可以包含新类型的项。
- 创建一个新的集合类，它可以包含新类型的项，实现所有需要的方法。

一般情况下，新的类型需要额外功能，所以常常需要用到新的集合类，因此创建集合类会花费大量时间。

另一方面，泛型类大大简化了这个问题。泛型类是以实例化过程中提供的类型或类为基础建立的，可以毫不费力地对对象进行强类型化。对于集合，创建“T类型对象的集合”十分简单，只需要编写一行代码即

可。不使用下面的代码：

```
CollectionClass items = new CollectionClass();  
items.Add(new ItemClass());
```

而是使用：

```
CollectionClass<ItemClass> items = new CollectionClass<ItemCla
```

```
items.Add(new ItemClass());
```

尖括号语法是把类型参数传送给泛型类型的方式。在上面的代码中，应把CollectionClass<ItemClass>看成ItemClass的CollectionClass。当然，本章后面会详细探讨这个语法。

泛型不只涉及集合，但是集合特别适合使用泛型。本章在后面介绍System.Collections.Generic名称空间时会看到这一点。创建一个泛型类，就可以生成一些方法，它们的签名可以强类型化为我们需要的任何类型，该类型甚至可以是值类型或引用类型，处理各自的操作。还可以把用于实例化泛型类的类型限制为支持某个给定的接口，或派生自某种类型，从而只允许使用类型的一个子集。泛型并不限于类，还可以创建泛型接口、泛型方法（可以在非泛型类上定义），甚至泛型委托。这将极大地提高代码的灵活性，正确使用泛型可以显著缩短开发时间。

注意： 对于熟悉C++或对C++感兴趣的读者来说，这是C++模板

和C#泛型类的一个区别。在C++中，编译器可以检测出在哪里使用了模板的某个特定类型，例如，模板B的A类型，然后编译需要的代码，来创建这个类型。而在C#中，所有操作都在运行期间进行。

那么该如何实现泛型呢？通常，在创建类时，它会编译为一个类型，然后在代码中使用。读者可能认为，在创建泛型类时，它只有被编译为许多类型，才能进行实例化。幸好并不是这样：在.NET中，类有无限多个。在后台，.NET运行库允许在需要时动态生成泛型类。在实例化之前，B的某个泛型类A甚至不存在。

12.2 使用泛型

在探讨如何创建自己的泛型类型之前，首先介绍.NET Framework提供的泛型，包括System. Collections.Generic名称空间中的类型，这个名称空间已在前面的代码中出现过多次，因为默认情况下它包含在控制台应用程序中。我们还没有使用过这个名称空间中的类型，但下面就要使用了。本节将讨论这个名称空间中的类型，以及如何使用它们创建强类型化的集合，改进已有集合的功能。

首先论述另一个较简单的泛型类型，即可空类型（nullable type），它解决了值类型的一个小问题。

12.2.1 可空类型

前面的章节介绍了值类型（大多数基本类型，例如，int、double和所有结构）区别于引用类型（string和任意类）的一种方式：值类型必须包含一个值，它们可以在声明之后、赋值之前，在未赋值的状态下存在，但不能使用未赋值的变量。而引用类型可以是null。

有时让值类型为空是很有用的（尤其是处理数据库时），泛型使用System.Nullable<T>类型提供了使值类型为空的一种方式。例如：

```
System.Nullable<int> nullableInt;
```

这行代码声明了一个变量nullableInt，它可以拥有int变量能包含的

任意值，还可以拥有值`null`。所以可以编写如下的代码：

```
nullableInt = null;
```

如果`nullableInt`是一个`int`类型的变量，上面的代码是不能编译的。

前面的赋值等价于：

```
nullableInt = new System.Nullable<int>();
```

与其他任意变量一样，无论是初始化为`null`（使用上面的语法），还是通过给它赋值来初始化，都不能在初始化之前使用它。

可以像测试引用类型一样测试可空类型，看看它们是否为`null`：

```
if (nullableInt == null)
{
    ...
}
```

另外，可以使用`HasValue`属性：

```
if (nullableInt.HasValue

)
{
    ...
}
```

这不适用于引用类型，即使引用类型有一个HasValue属性，也不能使用这种方法，因为引用类型的变量值为null，就表示不存在对象，当然就不能通过对象来访问这个属性，否则会抛出一个异常。

可使用Value属性来查看可空类型的值。如果HasValue是true，就说明Value属性有一个非空值。但如果HasValue是false，就说明变量被赋予了null，访问Value属性会抛出System.InvalidOperationException类型的异常。

可空类型非常有用，以至于修改了C#语法。声明可空类型的变量不使用上述语法，而是使用下面的语法：

```
int? nullableInt;
```

int?是System.Nullable<int>的缩写，但更便于读取。在后面的章节中就使用了这个语法。

1. 运算符和可空类型

对于简单类型（如int），可以使用+、-等运算符来处理值。而对于对应的可空类型，这是没有区别的：包含在可空类型中的值会隐式转换为需要的类型，使用适当的运算符。这也适用于结构和自己提供的运算符。例如：

```
int? op1 = 5;  
int? result = op1 * 2;
```

注意，其中result变量的类型也是int?。下面的代码不会被编译：


```
int? op1 = 5;  
int result = op1 * 2;
```

为使上面的代码正常工作，必须进行显式转换：

```
int? op1 = 5;  
int result = (int)op1 * 2;
```

或通过Value属性访问值：

```
int? op1 = 5;  
int result = op1.Value * 2;
```

只要op1有一个值，上面的代码就可以正常运行。如果op1是null，就会生成System.InvalidOperationException类型的异常。

这就引出了下一个问题：当运算表达式中的一个或两个值是null时，例如，下面代码中的op1，会发生什么情况？

```
int? op1 = null;
```

```
int? op2 = 5;
int? result = op1 * op2;
```

答案是：对于除了bool?外的所有简单可空类型，该操作的结果是null，可以把它解释为“不能计算”。对于结构，可以定义自己的运算符来处理这种情况（详见本章后面的内容）。对于bool?，为&和|定义的运算符会得到非空返回值，这些运算符的结果十分符合逻辑，如果不需要知道其中一个操作数的值，就可以计算出结果，则该操作数是否为null就不重要。

2. ??运算符

为进一步减少处理可空类型所需的代码量，使可空变量的处理变得更简单，可以使用??运算符。这个运算符称为空接合运算符（null coalescing operator），是一个二元运算符，允许给可能等于null的表达式提供另一个值。如果第一个操作数不是null，该运算符就等于第一个操作数，否则，该运算符就等于第二个操作数。下面的两个表达式的作用是相同的：

```
op1 ?? op2
op1 == null ? op2 : op1
```

在这两行代码中，op1可以是任意可空表达式，包括引用类型和更重要的可空类型。因此，如果可空类型是null，就可以使用??运算符提供要使用的默认值，如下所示：

```
int? op1 = null;
int result = op1 * 2 ?? 5;
```

在这个示例中，`op1`是`null`，所以`op1*2`也是`null`。但是，`??`运算符检测到这个情况，并把值5赋予`result`。这里要特别注意，在结果中放入`int`类型的变量`result`不需要显式转换。`??`运算符会自动处理这个转换。还可以把`??`表达式的结果放在`int?`中：

```
int? result = op1 * 2 ?? 5;
```

在处理可空变量时，`??`运算符有许多用途，它也是一种提供默认值的便捷方式，不需要使用`if`结构中的代码块或容易引起混淆的三元运算符。

3. ?.运算符

这个操作符通常称为`Elvis` 运算符或空条件运算符，有助于避免繁杂的空值检查造成的代码歧义。例如，如果想得到给定客户的订单数，就需要在设置计数值之前检查空值：

```
int count = 0;
if (customer.orders != null)
{
    count = customer.orders.Count();
}
```

如果只编写了这段代码，但客户没有订单（即是`null`），就会抛出`System.ArgumentNullException`：

```
int count = customer.orders.Count();
```

使用?.运算符，会把int? count设置为null，而不是抛出一个异常。

```
int? count = customer.orders?.Count();
```

结合前一节讨论的空合并操作符??与空条件运算符?.可以在结果是null时设置一个默认值。

```
int? count = customer.orders?.Count() ?? 0;
```

空条件运算符的另一个用途是触发事件。第13章详细讨论了事件。触发事件的最常见方法是使用如下代码模式：

```
var onChanged = OnChanged;  
if (onChanged != null)  
{  
    onChanged(this, args);  
}
```

这种模式不是线程安全的，因为有人会在null检查已经完成后，退订最后一个事件处理程序。此时会抛出异常，程序崩溃。使用空条件运算符可以避免这种情形：

```
OnChanged?.Invoke(this, args);
```

注意： 如果使用运算符重载方法（例如,==），但没有检查null，就会抛出System.NullReferenceException。

如第11章所述，在C:\BegVCSharp\Chapter12\Ch12CardLib\Card.cs类的==运算符重载中，使用?.运算符检查null，可以避免在使用该方法时抛出异常。例如：

```
public static bool operator ==(Card card1, Card card2)
    => (card1?.suit == card2?.suit) && (card1?.rank == car
```

在语句中包括空条件运算符，就清楚地表示：如果左边的对象（在本例中是card1或card2）不为空，就检索右边的对象。如果左边的对象为空（即card1或card2），就终止访问链，返回null。

4. 使用可空类型

在下面的示例中，将介绍可空类型Vector。

试一试：可空类型：**Ch12Ex01**

（1）在C:\BegVCSharp\Chapter12目录中创建一个新控制台应用程序项目Ch12Ex01。

（2）在文件Vector.cs中添加一个新类Vector。

（3）修改Vector.cs中的代码，如下所示：

```
using static System.Math;
```

public

class Vector

{

public double? R = null;

public double? Theta = null;

public double? ThetaRadians

{

// Convert degrees to radians.

```
get { return (Theta * Math.PI / 180.0); }  
  
}
```

```
public Vector(double? r, double? theta)
```

```
{
```

```
    // Normalize.
```

```
    if (r<0)
```

```
{
```

```
    r = -r;
```

```
    theta += 180;
```

```
}
```

```
theta = theta % 360;
```

```
// Assign fields.
```

```
R = r;
```



```
Theta = theta;
```

```
}
```

```
public static Vector operator +(Vector op1, Vector op2)
```

```
{
```

```
try
```

```
{
```

```
// Get (x, y) coordinates for new vector.
```

```
double newX = op1.R.Value * Sin(op1.ThetaRadians.Value)
```

```
+ op2.R.Value * Sin(op2.ThetaRadians.Value);
```

```
double newY = op1.R.Value * Cos(op1.ThetaRadians.Value)
```

```
+ op2.R.Value * Cos(op2.ThetaRadians.Value);
```

```
// Convert to (r, theta).
```

```
double newR = Sqrt(newX * newX + newY * newY);
```

```
double newTheta = Atan2(newX, newY) * 180.0 / PI;
```

```
// Return result.
```

```
return new Vector(newR, newTheta);
```

```
}
```

```
catch
```

```
{
```

```
// Return "null" vector.
```

```
return new Vector(null, null);
```

```
}
```

```
}
```

```
public static Vector operator -(Vector op1) => new Vector(-o
```

```
public static Vector operator -(Vector op1, Vector op2) => c
```

```
public override string ToString()
```

```
{
```

```
// Get string representation of coordinates.
```

```
string rString = R.HasValue ? R.ToString(): "null";
```

```
string thetaString = Theta.HasValue ? Theta.ToString(): "n
```

```
// Return (r, theta) string.
```

```
return string.Format("${rString}, {thetaString}");
```

```
}
```

```
}
```

(4) 修改Program.cs中的代码，如下所示：

```
class Program
{
    static void Main(string[] args)
    {
        Vector v1 = GetVector("vector1");

        Vector v2 = GetVector("vector1");

        WriteLine($"{v1} + {v2} = {v1 + v2}");

        WriteLine($"{v1} - { v2} = {v1 - v2}");

        ReadKey();
    }
}
```

```
static Vector GetVector(string name)
```

```
{
```

```
    WriteLine($"Input {name} magnitude:");
```

```
    double? r = GetNullableDouble();
```

```
    WriteLine($"Input {name} angle (in degrees):");
```

```
    double? theta = GetNullableDouble();
```

```
return new Vector(r, theta);
```

```
}
```

```
static double? GetNullableDouble()
```

```
{
```

```
double? result;
```

```
string userInput = ReadLine();
```

```
try
```



```
{
```

```
    result = double.Parse(userInput);
```

```
}
```

```
catch
```

```
{
```

```
    result = null;
```

```
}
```

```

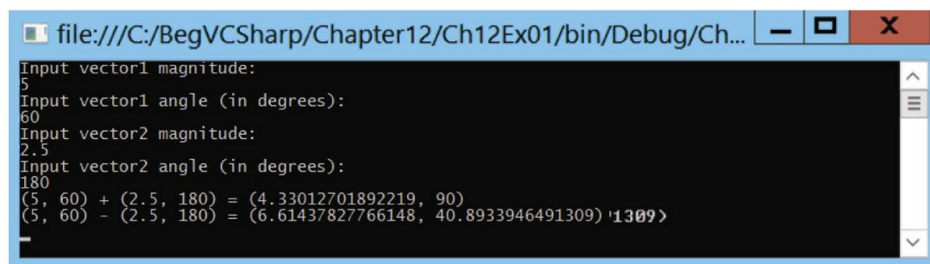
    return result;

}

}

```

（5）执行应用程序，给两个矢量（vector）输入值，示例输出结果如图12-1所示。



```

file:///C:/BegVCSharp/Chapter12/Ch12Ex01/bin/Debug/Ch...
Input vector1 magnitude:
5
Input vector1 angle (in degrees):
60
Input vector2 magnitude:
2.5
Input vector2 angle (in degrees):
180
(5, 60) + (2.5, 180) = (4.33012701892219, 90)
(5, 60) - (2.5, 180) = (6.61437827766148, 40.8933946491309) '1309'

```

图12-1

（6）再次执行应用程序，这次跳过四个值中的至少一个，示例输出结果如图12-2所示。

```
file:///C:/BegVCSharp/Chapter12/Ch12Ex01/bin/Debug/Ch...
Input vector1 magnitude:
5
Input vector1 angle <in degrees>:
60
Input vector1 magnitude:
100
Input vector1 angle <in degrees>:
180
<5, 60> + <null, 180> = <null, null>
<5, 60> - <null, 180> = <null, null>
```

图12-2

示例的说明

在这个示例中，创建了一个类Vector，它表示带极坐标（有一个幅值和一个角度）的矢量，如图12-3所示。

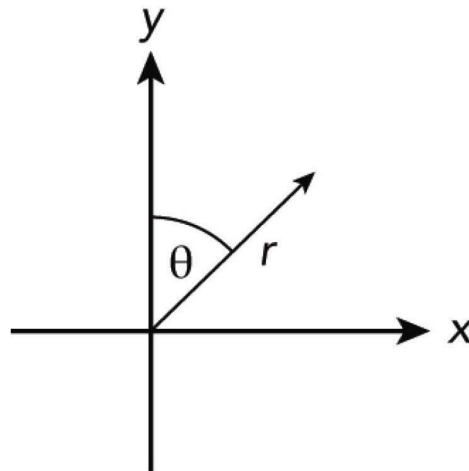


图12-3

坐标 r 和 θ 在代码中用公共字段 R 和 Θ 表示，其中 Θ 的单位是度（ $^{\circ}$ ）。 Θ Radians用于获取 Θ 的弧度值，这是必需的，因为Math类在其静态方法中使用弧度。 R 和 Θ 的类型都是double?，所以它们可以为空。

```
public class Vector
```

```

{
    public double? R = null;
    public double? Theta = null;
    public double? ThetaRadians
    {
        get
        {
            // Convert degrees to radians.
            return (Theta * PI / 180.0);
        }
    }
}

```

Vector的构造函数标准化R和Theta的初始值，然后赋予公共字段。

```

public Vector(double? r, double? theta)
{
    // Normalize.
    if (r<0)
    {
        r = -r;
        theta += 180;
    }
    theta = theta % 360;
    // Assign fields.
    R = r;
    Theta = theta;
}

```

Vector类的主要功能是使用运算符重载对矢量进行相加和相减运算，这需要一些非常基本的三角函数知识，这里不解释它们，相关内容可以访问站点<http://www.onlinemathlearning.com/basic-trigonometry.html>，或者在互联网上搜索其他资源。在代码中，重要的是，如果在获取R或ThetaRadians的Value属性时抛出了异常，即其中一个为null，就返回“空”矢量。

```
public static Vector operator +(Vector op1, Vector op2)
{
    try
    {
        // Get (x, y) coordinates for new vector.
        ...
    }
    catch
    {
        // Return "null" vector.
        return new Vector(null, null);
    }
}
```

如果组成矢量的一个坐标是null，该矢量就是无效的，这里用R和Theta都可为null的Vector类来表示。Vector类的其他代码重写了其他运算符，以便扩展相加的功能，使其包含相减操作，再重写ToString()，

获取Vector对象的字符串表示。

Program.cs中的代码测试Vector类，让用户初始化两个矢量，再对它们进行相加和相减。如果用户省略了某个值，该值就解释为null，应用前面提及的规则。

12.2.2 System.Collections.Generic名称空间

实际上，本书前面的每个应用程序都包含如下名称空间：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

System名称空间包含.NET应用程序使用的大多数基本类型。System.Text名称空间包含与字符串处理和编码相关的类型，System.Linq名称空间将在本书后面介绍。System.Threading.Tasks名称空间包含帮助编写异步代码的类型，本书不予讨论。但System.Collections.Generic名称空间包含什么类型？为什么要在默认情况下把它包含在控制台应用程序中？

这个名称空间包含用于处理集合的泛型类型，使用得非常频繁，所以用using语句配置它，这样使用时就不必添加限定符了。

下面介绍这些泛型类型，它们可以使工作更容易完成，可以毫不费力地创建强类型化的集合类。表12-1描述了本节要介绍的System.Collections.Generic名称空间中的两个类型，本章后面还会详细阐述这个名称空间中的更多类型。

表12-1 泛型集合类型

类型	说明
List<T>	T类型对象的集合
Dictionary<K, V>	与K类型的键值相关的V类型的项的集合

后面还会介绍和这些类一起使用的各种接口和委托。

1. List<T>

List<T>泛型集合类型更加快捷、更便于使用；这样，就不必像上一章那样，从CollectionBase中派生一个类，然后实现需要的方法。它的另一个好处是正常情况下需要实现的许多方法（例如，Add()）已经自动实现了。

创建T类型对象的集合需要如下代码：

```
List<T> myCollection = new List<T>();
```

这就足够了。未必要定义类、实现方法或执行其他操作。还可以把List<T>对象传送给构造函数，在集合中设置项的起始列表。List<T>还

有一个Item属性，允许进行类似于数组的访问，如下所示：

```
T itemAtIndex2 = myCollectionOfT[2];
```

这个类还支持其他几个方法，但只要掌握了上述知识，就完全可以开始使用该类了。下面的示例将介绍如何实际使用List<T>。

试一试：使用List<T>： Ch12Ex02

（1）在C:\BegVCSharp\Chapter12目录中创建一个新控制台应用程序Ch12Ex02。

（2）在Solution Explorer窗口中右击项目名称，选择Add|Existing Item选项。

（3）在C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01目录中选择Animal.cs、Cow.cs和Chicken.cs文件，单击Add按钮。

（4）修改这3个文件中的名称空间声明，如下所示：

```
namespace Ch12Ex02
```

（5）修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)
{
    List<Animal> animalCollection = new List<Animal>();
```



```
animalCollection.Add(new Cow("Rual"));
```

```
animalCollection.Add(new Chicken("Donna"));
```

```
foreach (Animal myAnimal in animalCollection)
```

```
{
```

```
    myAnimal.Feed();
```

```
}
```

```
ReadKey();
```

```
}
```

(6) 执行应用程序，结果与第11章的Ch11Ex02的结果相同。

示例的说明

这个示例与Ch11Ex02只有两个区别。第一个区别是下面的代码行：

```
Animals animalCollection = new Animals();
```

被替换为：

```
List<Animal> animalCollection = new List<Animal>();
```

第二个区别比较重要：项目中不再有Animals集合类。通过使用泛型的集合类，前面为创建这个类所做的工作现在用一行代码即可完成。

获得相同效果的另一个方法是不修改Program.cs中的代码，而是使用Animals的如下定义：

```
public class Animals : List<Animal> {}
```

这么做的优点是，能较容易地看懂Program.cs中的代码，还可以在合适时给Animals类添加成员。

2. 对泛型列表进行排序和搜索

对泛型列表进行排序与对其他列表进行排序是一样的。第11章介绍了如何使用IComparer和Comparable接口比较两个对象，然后对该类型的对象列表排序。这里唯一的区别在于，可以使用泛型接口IComparer<T>和Comparable<T>，它们提供了略有区别的、针对特定类型的方法。表12-2列出了它们之间的区别。

表12-2 对泛型列表进行排序

泛型方法	非泛型方法	区别
int Comparable<T>. CompareTo (T otherObj)	int Comparable. CompareTo (object otherObj)	在泛型版本中是强类型化的
bool Comparable<T>. Equals (T otherObj)	N/A	在非泛型接口中不存在，可以改用继承的 object.Equals ()
int IComparer<T>.Compare (T objectA, T objectB)	int IComparer. Compare (object objectA, object objectB)	在泛型版本中是强类型化的
bool IComparer<T>.Equals (T objectA, T objectB)	N/A	在非泛型接口中不存在，可以改用继承的 object.Equals ()
int IComparer<T>. GetHashCode (T		在非泛型接口中不存在，可以改用继

objectA)	N/A	承的object. GetHashCode ()
----------	-----	------------------------------

要对List<T>排序，可以在要排序的类型上提供IComparable<T>接口，或者提供IComparer<T>接口。另外，还可以提供泛型委托，作为排序方法。从了解代码工作原理的角度看，这非常有趣，因为实现上述接口并不比实现其非泛型版本更麻烦。

一般情况下，给列表排序需要有一个方法来比较两个T类型的对象。要在列表中搜索，只需要一个方法来检查T类型的对象，看看它是否满足某个条件。定义这样的方法很简单，这里给出两个可以使用的泛型委托类型：

- **Comparison<T>**：这个委托类型用于排序方法，其返回类型和参数如下：

```
int method(T objectA, T objectB)
```

- **Predicate<T>**：这个委托类型用于搜索方法，其返回类型和参数如下：

```
bool method(T targetObject)
```

可以定义任意多个这样的方法，使用它们实现List<T>的搜索和排序方法。下面的示例进行了演示。

试一试：**List<T>的搜索和排序**： **Ch12Ex03**

(1) 在C:\BegVCSharp\Chapter12目录中创建一个新控制台应用程序Ch12Ex03。

(2) 在Solution Explorer窗口中右击项目名称，然后选择Add Existing Item选项。

(3) 在C:\BegVCSharp\Chapter12\Ch12Ex01\Ch12Ex01目录中选择Vector.cs文件，单击Add按钮。

(4) 修改这个文件中的名称空间声明，如下所示：

```
namespace Ch12Ex03
```

(5) 添加一个新类Vectors。

(6) 修改Vectors.cs中的代码，如下所示：

```
public
```

```
class Vectors : List<Vector>
```

```
{
```

```
    public Vectors()
```

```

{
}
public Vectors(IEnumerable<Vector> initialItems)
{
    foreach (Vector vector in initialItems)
    {
        Add(vector);
    }
}
public string Sum()
{
    StringBuilder sb = new StringBuilder();
    Vector currentPoint = new Vector(0.0, 0.0);
    sb.Append("origin");
    foreach (Vector vector in this)
    {
        sb.AppendFormat($" + {vector}");
        currentPoint += vector;
    }
    sb.AppendFormat($" = {currentPoint}");
    return sb.ToString();
}
}

```

(7) 添加一个新类VectorDelegates。

(8) 修改VectorDelegates.cs中的代码，如下所示：

public static

class VectorDelegates

{

public static int Compare(Vector x, Vector y)

{

if (x.R > y.R)

{

return 1;

}

```
else if (x.R<y.R)
```

```
{
```

```
    return -1;
```

```
}
```

```
return 0;
```

```
}
```

```
public static bool TopRightQuadrant(Vector target)
```



```
{
```

```
    if (target.Theta >= 0.0 && target.Theta<= 90.0)
```

```
    {
```

```
        return true;
```

```
    }
```

```
else
```

```
{  
  
    return false;  
  
}  
  
}  
  
}
```

(9) 修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)  
{  
    Vectors route = new Vectors();  
  
    route.Add(new Vector(2.0, 90.0));  
}
```

```
route.Add(new Vector(1.0, 180.0));
```

```
route.Add(new Vector(0.5, 45.0));
```

```
route.Add(new Vector(2.5, 315.0));
```

```
WriteLine(route.Sum());
```

```
Comparison<Vector> sorter = new Comparison<Vector>(
```

```
VectorDelegates.Compare);
```

```
route.Sort(sorter);
```

```
WriteLine(route.Sum());
```

```
Predicate<Vector> searcher =
```

```
new Predicate<Vector>(VectorDelegates.TopRightQuadrant);
```

```
Vectors topRightQuadrantRoute = new Vectors(route.FindAll(
```

```
WriteLine(topRightQuadrantRoute.Sum());
```

```
ReadKey();
```

}

(10) 执行应用程序，结果如图12-4所示。

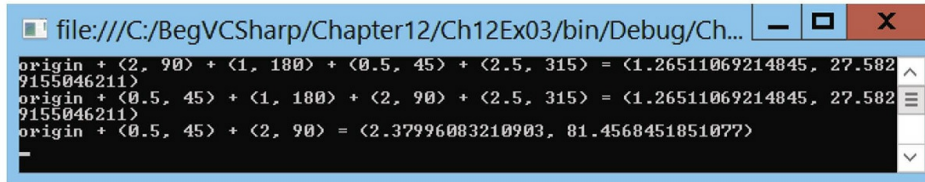


图12-4

示例的说明

在这个示例中，为Ch12Ex01中的Vector类创建了一个集合类Vectors。可以只使用List<Vector>类型的变量，但因为需要其他功能，所以使用了一个新类Vectors，它派生自List<Vector>，允许添加需要的其他成员。

该类有一个返回字符串的成员Sum()，该字符串依次列出每个矢量，以及把它们加在一起（使用源类Vector的重载+运算符）的结果。每个矢量都可以看成“方向+距离”，所以这个矢量列表构成了一条有端点的路径。

```
public string Sum()
{
    StringBuilder sb = new StringBuilder();
    Vector currentPoint = new Vector(0.0, 0.0);
    sb.Append("origin");
    foreach (Vector vector in this)
```

```

{
    sb.AppendFormat(" + {vector}");
    currentPoint += vector;
}
sb.AppendFormat(" = {currentPoint}");
return sb.ToString();
}

```

该方法使用System.Text名称空间中简便的StringBuilder类来构建响应字符串。这个类包含这里使用的Append()和AppendFormat()等成员，所以很容易组合字符串，其性能也比串联各个字符串要高。使用这个类的ToString()方法即可获得最终字符串。

本例还创建了两个用作委托的方法，作为VectorDelegates的静态成员。Compare()用于比较（排序），TopRightQuadrant()用于搜索。稍后在分析Program.cs中的代码时介绍它们。

Main()中的代码首先初始化Vectors集合，给它添加几个Vector对象（这段代码包含在Ch12Ex03\Program.cs文件中）：

```

Vectors route = new Vectors();
route.Add(new Vector(2.0, 90.0));
route.Add(new Vector(1.0, 180.0));
route.Add(new Vector(0.5, 45.0));
route.Add(new Vector(2.5, 315.0));

```

如前所述，Vectors.Sum()方法用于输出集合中的项，这次是按照其初始顺序输出：

```
WriteLine(route.Sum());
```

接着创建第一个委托sorter，这个委托属于Comparison<Vector>类型，因此可以赋予带如下返回类型和参数的方法：

```
int method
```

```
(Vector objectA, Vector objectB)
```

它匹配VectorDelegates.Compare()，该方法就是赋予委托的方法。

```
Comparison<Vector> sorter = new Comparison<Vector>(
    VectorDelegates.Compare);
```

Compare()比较两个矢量的大小，如下所示：

s

```
public static int Compare(Vector x, Vector y)
{
    if (x.R > y.R)
    {
        return 1;
    }
    else if (x.R < y.R)
    {
        return -1;
    }
    return 0;
}
```

```
}
```

这样就可以按大小对矢量排序了：

```
route.Sort(sorter);  
WriteLine(route.Sum());
```

应用程序的输出结果符合我们的预期——汇总的结果是一样的，因为无论用什么顺序执行各个步骤，“矢量路径”的端点都是相同的。

然后进行搜索，获取集合中的一个矢量子集。这需要使用 `VectorDelegates.TopRightQuadrant()` 来实现：

```
public static bool TopRightQuadrant(Vector target)  
{  
    if (target.Theta >= 0.0 && target.Theta <= 90.0)  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

如果方法的 `Vector` 参数的 `Theta` 的值介于 $0^\circ \sim 90^\circ$ 之间，该方法就返回 `true`，也就是说，它在前面的排序图中指向上或右。

在 `Main()` 方法中，通过 `Predicate<Vector>` 类型的委托使用这个方

法，如下所示：

```
Predicate<Vector> searcher =  
    new Predicate<Vector>(VectorDelegates.TopRightQuadrant);  
Vectors topRightQuadrantRoute = new Vectors(route.FindAll(s  
WriteLine(topRightQuadrantRoute.Sum()));
```

这需要在Vectors中定义构造函数：

```
public Vectors(IEnumerable<Vector> initialItems)  
{  
    foreach (Vector vector in initialItems)  
    {  
        Add(vector);  
    }  
}
```

其中，使用IEnumerable<Vector>的接口初始化了一个新的Vectors集合，这是必需的，因为List<Vector>.FindAll()返回一个List<Vector>实例，而不是Vectors实例。

搜索的结果是，只返回Vector对象的一个子集，所以汇总的结果不同（这正是我们希望的）。使用这些泛型委托类型来排序和搜索泛型集合需要一段时间才能习惯，但代码更加流畅高效了，代码的结构更富逻辑性。最好花点时间研究本节介绍的技术。

另外，在这个示例中，注意下面的代码：

```
Comparison<Vector> sorter = new Comparison<Vector>(
```

```
VectorDelegates.Compare);  
route.Sort(sorter);
```

可以简化为:

```
route.Sort(VectorDelegates.Compare);
```

这样就不需要隐式引用`Comparison<Vector>`类型了。实际上，仍会创建这个类型的一个实例，但它是隐式创建的。显然，`Sort()`方法需要这个类型的实例才能工作，但编译器会认识到这一点，在我们提供的方法中自动创建该类型的实例。此时，对`VectorDelegates.Compare()`的引用（没有括号）称为方法组。许多情况下，都可以使用方法组以这种方式隐式地创建委托，使代码变得更便于读取。

3. Dictionary<K, V>

这个类型可定义键/值对的集合。与本章前面介绍的其他泛型集合类型不同，这个类需要实例化两个类型，分别用于键和值，以表示集合中的各个项。

实例化`Dictionary<K, V>`对象后，就可以像在继承自`DictionaryBase`的类上那样，对它执行相同的操作，但要使用已有的类型安全的方法和属性。例如，使用强类型化的`Add()`方法添加键/值对。

```
Dictionary<string, int> things = new Dictionary<string, int>()  
things.Add("Green Things", 29);  
things.Add("Blue Things", 94);
```

```
things.Add("Yellow Things", 34);  
things.Add("Red Things", 52);  
things.Add("Brown Things", 27);
```

可使用Keys和Values属性迭代集合中的键和值：

```
foreach (string key in things.Keys)  
{  
    WriteLine(key);  
}  
foreach (int value in things.Values)  
{  
    WriteLine(value);  
}
```

还可以迭代集合中的各个项，把每个项作为一个KeyValuePair<K, V>实例来获取，这与第11章介绍的DictionaryEntry对象十分相似：

```
foreach (KeyValuePair<string, int> thing in things)  
{  
    WriteLine($"{thing.Key} = {thing.Value}");  
}
```

对于Dictionary<K, V>要注意的一点是，每个项的键都必须是唯一的。如果要添加的项的键与已有项的键相同，就会抛出ArgumentOutOfRangeException异常。所以，Dictionary<K, V>允许把IComparer<K>接口传递给其构造函数。如果要把自己的类用作键，且它们不支持IComparable或IComparable<K>接口，或者要使用非默认的过程比较对

象，就必须把IComparer<K>接口传递给其构造函数。例如，在上面的示例中，可以使用不区分大小写的方法来比较字符串键：

```
Dictionary<string, int> things =  
    new Dictionary<string, int>(StringComparer.CurrentCultureIg
```

如果使用下面的键，就会得到一个异常：

```
things.Add("Green Things", 29);  
things.Add("Green things", 94);
```

也可以给构造函数传递初始容量（使用int）或项的集合（使用IDictionary<K,V>接口）。

C# 6引入了一个新特性：索引初始化器，它支持在对象初始化器内部初始化索引：

```
var zahlen = new Dictionary<int, string>()  
{  
    [1] = "eins",  
    [2] = "zwei"  
};
```

索引初始化器很方便，因为在许多情况下都不需要通过var zahlen显示临时变量。使用表达式体方法，上面的例子会级联简化的作用：

```
public ZObject ToGerman() => new ZObject() { [1] = "eins", [2]
```

4. 修改CardLib以便使用泛型集合类

对前几章创建的CardLib项目可以进行简单的修改，即修改Cards集合类，以使用一个泛型集合类，这将减少许多行代码。对Cards的类定义需要做如下修改（这段代码包含在Ch12CardLib\Cards.cs文件中）：

```
public class Cards : List<Card>

, ICloneable { ... }
```

还可以删除Cards的所有方法，但Clone()和CopyTo()除外，因为Clone()是ICloneable需要的方法，而List<Card>提供的CopyTo()版本处理的是Card对象数组，而不是Cards集合。需要对Clone()做一些轻微的修改，因为List<T>没有定义List属性：

```
public object Clone()
{
    Cards newCards = new Cards();
    foreach (Card sourceCard in this

)
    {
        newCards.Add((Card)sourceCard.Clone());
    }
    return newCards;
}
```

这里没有列出代码，因为这是十分简单的修改，CardLib的更新版

本为Ch12CardLib，它与第11章的客户代码包含在本章的下载代码中。

12.3 定义泛型类型

利用前面介绍的泛型知识，足以创建自己的泛型了。前面的许多代码都涉及到泛型类型，还看到了多个使用泛型语法的实例。本节将定义如下内容：

- 泛型类
- 泛型接口
- 泛型方法
- 泛型委托

在定义泛型类型的过程中，还将讨论下面一些更高级的技术：

- default关键字
- 约束类型
- 从泛型类中继承
- 泛型运算符

12.3.1 定义泛型类

要创建泛型类，只需在类定义中包含尖括号语法：

```
class MyGenericClass<T> { ... }
```

其中T可以是任意标识符，只要遵循通常的C#命名规则即可，例如，不以数字开头等。但一般只使用T。泛型类可在其定义中包含任意

多个类型参数，参数之间用逗号分隔开，例如：

```
class MyGenericClass<T1, T2, T3
```

```
> { ... }
```

定义了这些类型后，就可以在类定义中像使用其他类型那样使用它们。可以把它们用作成员变量的类型、属性或方法等成员的返回类型以及方法的参数类型等。例如：

```
class MyGenericClass<T1, T2, T3>
{
    private T1 innerT1Object;

    public MyGenericClass(T1 item)

    {

        innerT1Object = item;
```



```
}
```

```
public T1 InnerT1Object
```

```
{
```

```
get { return innerT1Object; }
```

```
}
```

```
}
```

其中，类型T1的对象可以传递给构造函数，只能通过InnerT1Object属性对这个对象进行只读访问。注意，不能假定为类提供了什么类型。例如，下面的代码就不会编译：

```
class MyGenericClass<T1, T2, T3>
{
    private T1 innerT1Object;
    public MyGenericClass()

    {

        innerT1Object = new T1();

    }

    public T1 InnerT1Object
    {
        get { return innerT1Object; }
    }
}
```

我们不知道T1是什么，也就不能使用它的构造函数，它甚至可能没有构造函数，或者没有可公共访问的默认构造函数。如果不使用涉及本

节后面介绍的高级技术的复杂代码，则只能对T1进行如下假设：可以把它看成继承自System.Object的类型或可以封箱到System.Object中的类型。

显然，这意味着不能对这个类型的实例进行非常有趣的操作，或者对为MyGenericClass泛型类提供的其他类型进行有趣的操作。不使用反射（这是用于在运行期间检查类型的高级技术，本章不介绍它），就只能使用下面的代码：

```
public string GetAllTypesAsString()
{
    return "T1 = " + typeof(T1).ToString()
        + ", T2 = " + typeof(T2).ToString()
        + ", T3 = " + typeof(T3).ToString();
}
```

可以做一些其他工作，尤其是对集合进行操作，因为处理对象组是非常简单的，不需要对对象类型进行任何假设，这是为什么存在本章前面介绍的泛型集合类的一个原因。

另一个需要注意的限制是，在比较为泛型类型提供的类型值和null时，只能使用运算符==和!=。例如，下面的代码会正常工作：

```
public bool Compare(T1 op1, T1 op2)
{
    if (op1 != null && op2 != null)
```

```

    {
        return true;
    }
    else
    {
        return false;
    }
}

```

其中，如果T1是一个值类型，则总是假定它是非空的，于是在上面的代码中，Compare总是返回true。但是，下面试图比较两个实参op1和op2的代码将不能编译：

```

public bool Compare(T1 op1, T1 op2)
{
    if (op1 == op2)

    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```
}
```

其原因是这段代码假定T1支持==运算符。这说明，要对泛型进行实际的操作，需要更多地了解类中使用的类型。

1. default关键字

要确定用于创建泛型类实例的类型，需要了解一个最基本的情况：它们是引用类型还是值类型。若不知道这个情况，就不能用下面的代码赋予null值：

```
public MyGenericClass()  
{  
    innerT1Object = null;  
}
```

如果T1是值类型，则innerT1Object不能取null值，所以这段代码不会编译。幸好，开发人员考虑到了这个问题，使用default关键字（本书前面在switch结构中使用过它）的新用法解决了它。该新用法如下：

```
public MyGenericClass()  
{  
    innerT1Object = default(T1);  
  
}
```

其结果是，如果`innerT1Object`是引用类型，就给它赋予`null`值；如果它是值类型，就给它赋予默认值。对于数字类型，这个默认值是`0`；而结构根据其各个成员的类型，以相同的方式初始化为`0`或`null`。`default`关键字允许对必须使用的类型执行更多操作，但为了更进一步，还需要限制所提供的类型。

2. 约束类型

前面用于泛型类的类型称为无绑定（`unbounded`）类型，因为没有对它们进行任何约束。而通过约束（`constraining`）类型，可以限制可用于实例化泛型类的类型，这有许多方式。例如，可以把类型限制为继承自某个类型。回顾前面使用的`Animal`、`Cow`和`Chicken`类，可以把一个类型限制为`Animal`或继承自`Animal`，则下面的代码是正确的：

```
MyGenericClass<Cow> = new MyGenericClass<Cow>();
```

但下面的代码不能编译：

```
MyGenericClass<string
```

```
> = new MyGenericClass<string
```

```
>();
```

在类定义中，这可以使用`where`关键字来实现：

```
class MyGenericClass<T> where T : constraint
```

```
{ ... }
```

其中*constraint*定义了约束。可以用这种方式提供许多约束，各个约束之间用逗号分开：

```
class MyGenericClass<T> where T :  
constraint1
```

```
,
```

```
constraint2
```

```
{ ... }
```

还可以使用多个*where*语句，定义泛型类需要的任意类型或所有类

型上的约束：

```
class MyGenericClass<T1, T2> where T1 : constraint1
```

```
where T2 : constraint2
```

```
{ ... }
```

约束必须出现在继承说明符的后面：

```
class MyGenericClass<T1, T2> : MyBaseClass
```

```
, IMyInterface
```

```
where T1 : constraint1 where T2 : constraint2 { ... }
```

表12-3中列出一些可用的约束。

表12-3 泛型类型约束

约束	定义	用法示例
<code>struct</code>	类型必须是值类型	在类中，需要值类型才能起作用，例如，T类型的成员变量是0，表示某种含义
<code>class</code>	类型必须是引用类型	在类中，需要引用类型才能起作用，例如，T类型的成员变量是null，表示某种含义
<i>base-class</i>	类型必须是基类或继承自基类。可以给这个约束提供任意类名	在类中，需要继承自基类的某种基本功能，才能起作用
<i>interface</i>	类型必须是接口或实现了接口	在类中，需要接口公开的某种基本功能，才能起作用
<code>new()</code>	类型必须有一个公共的无参构造函数	在类中，需要能实例化T类型的变量，例如在构造函数中实例化

注意： 如果new()用作约束，它就必须是为类型指定的最后一个约束。

可通过base-class约束，把一个类型参数用作另一个类型参数的约束，如下所示：

```
class MyGenericClass<T1, T2> where T2 : T1
```

```
{ ... }
```

其中，T2必须与T1的类型相同，或者继承自T1。这称为裸类型约束（naked type constraint），表示一个泛型类型参数用作另一个类型参数的约束。

类型约束不能循环，例如：

```
class MyGenericClass<T1, T2> where T2 : T1 where T1 : T2

{ ... }
```

这段代码不能编译。下面的示例将定义和使用一个泛型类，该类使用前面几章介绍的Animal类系列。

试一试：定义泛型类：Ch12Ex04

- （1）在C:\BegVCSharp\Chapter12目录中创建一个新的控制台应用程序Ch12Ex04。
- （2）在Solution Explorer窗口中右击项目名称，选择Add Existing Item选项。
- （3）从C:\BegVCSharp\Chapter12\Ch12Ex02\Ch12Ex02目录中选择Animal.cs、Cow.cs和Chicken.cs文件，单击Add按钮。
- （4）在已经添加的文件中修改名称空间声明，如下所示：

namespace **Ch12Ex04**

(5) 修改Animal.cs, 如下所示:

```
public abstract class Animal
{
    ...

    public abstract void MakeANoise();

}
```

(6) 修改Chicken.cs, 如下所示:

```
public class Chicken : Animal
{
    ...
    public override void MakeANoise()
```

```
{
```

```
    WriteLine($"{name} says 'cluck!';");
```

```
}
```

```
}
```

(7) 修改Cow.cs, 如下所示:

```
public class Cow : Animal
```

```
{
```

```
    ...
```

```
    public override void MakeANoise()
```

```
{
```

```
WriteLine($"{name} says 'moo!'");
```

```
}
```

```
}
```

（8）添加一个新类SuperCow，并修改SuperCow.cs中的代码，如下所示：

```
public
```

```
class SuperCow : Cow
```

```
{
```

```
    public void Fly()
```

```
{
```

```
WriteLine($"{name} is flying!");
```

```
}
```

```
public SuperCow(string newName): base(newName)
```

```
{
```

```
}
```

```
public override void MakeANoise()
```

```
{
```

```

        WriteLine(

            $"{name} says 'here I come to save the day!'" );

    }
}

```

（9）添加一个新类Farm，并修改Farm.cs中的代码，如下所示：

```

using System;
using System.Collections;

using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Ch12Ex04
{

```

```
public class Farm<T> : IEnumerable<T>
```

```
where T : Animal
```

```
{
```

```
private List<T> animals = new List<T>();
```

```
public List<T> Animals
```

```
{
```

```
get { return animals; }
```



```
}
```

```
public IEnumerator<T> GetEnumerator() => animals.GetEnumerator()
```

```
IEnumerator IEnumerable.GetEnumerator() => animals.GetEnumerator()
```

```
public void MakeNoises()
```

```
{
```

```
    foreach (T animal in animals)
```

```
    {
```

```
    animal.MakeANoise();
```

```
}
```

```
}
```

```
public void FeedTheAnimals()
```

```
{
```

```
    foreach (T animal in animals)
```

```
{
```

```
    animal.Feed();
```

```
}
```

```
}
```

```
public Farm<Cow> GetCows()
```

```
{
```

```
    Farm<Cow> cowFarm = new Farm<Cow>();
```

```
foreach (T animal in animals)
```

```
{
```

```
    if (animal is Cow)
```

```
    {
```

```
        cowFarm.Animals.Add(animal as Cow);
```

```
    }
```

```
}
```

```
return cowFarm;
```

```
}
```

```
}
```

```
}
```

(10) 修改Program.cs, 如下所示:

```
static void Main(string[] args)
```

```
{
```

```
    Farm<Animal> farm = new Farm<Animal>();
```

```
    farm.Animals.Add(new Cow("Rual"));
```

```
farm.Animals.Add(new Chicken("Donna"));
```

```
farm.Animals.Add(new Chicken("Mary"));
```

```
farm.Animals.Add(new SuperCow("Ben"));
```

```
farm.MakeNoises();
```

```
Farm<Cow> dairyFarm = farm.GetCows();
```

```
dairyFarm.FeedTheAnimals();
```

```
foreach (Cow cow in dairyFarm)
```

```
{
```

```
    if (cow is SuperCow)
```

```
    {
```

```
        (cow as SuperCow).Fly();
```

```
    }
```

```
}
```

```
ReadKey();
```

}

(11) 执行应用程序，结果如图12-5所示。

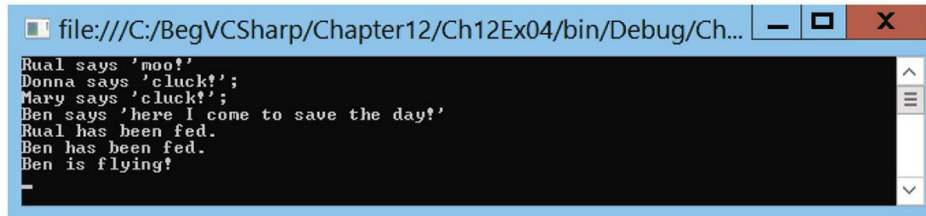


图12-5

示例的说明

在这个示例中，创建了一个泛型类Farm<T>，它没有继承泛型List类，而将泛型List类作为公共属性公开，该List的类型由传送给Farm<T>的类型参数T确定，且被约束为Animal，或者继承自Animal。

```
public class Farm<T> : IEnumerable<T>
    where T : Animal
{
    private List<T> animals = new List<T>();
    public List<T> Animals
    {
        get { return animals; }
    }
}
```

Farm<T>还实现了IEnumerable<T>，其中，T传递给这个泛型接

口，因此也以相同的方式进行了约束。实现这个接口，就可以迭代包含在Farm<T>中的项，而不必显式迭代Farm<T>.Animals。很容易就能做到这一点，只需要返回Animals公开的枚举器即可，该枚举器是一个List<T>类，也实现了IEnumerable<T>。

```
public IEnumerator<T> GetEnumerator() => animals.GetEnumerator()
```

因为IEnumerable<T>继承自IEnumerable，所以还需要实现IEnumerable.GetEnumerator()：

```
IEnumerator IEnumerable.GetEnumerator() => animals.GetEnumerator()
```

之后，Farm<T>包含的两个方法利用了抽象类Animal的方法：

```
public void MakeNoises()
{
    foreach (T animal in animals)
    {
        animal.MakeANoise();
    }
}

public void FeedTheAnimals()
{
    foreach (T animal in animals)
    {
        animal.Feed();
    }
}
```

T被约束为Animal，所以这段代码会正确编译——无论T实际上是什么，都可以访问MakeANoise()和Feed()方法。

下一个方法GetCows()更有趣。这个方法提取了集合中类型为Cow（或继承自Cow，例如，新的SuperCow类）的所有项：

```
public Farm<Cow> GetCows()
{
    Farm<Cow> cowFarm = new Farm<Cow>();
    foreach (T animal in animals)
    {
        if (animal is Cow)
        {
            cowFarm.Animals.Add(animal as Cow);
        }
    }
    return cowFarm;
}
```

有趣的是，这个方法似乎有点浪费。如果以后希望有同一系列的其他方法，如GetChickens()，也需要显式实现它们。在使用许多类型的系统中，需要更多方法。一个较好的解决方案是使用泛型方法，详见本章后面的内容。

Program.cs中的客户代码测试了Form的各个方法，它包含的代码大多已在前面列出，所以不需要深入探讨这些代码。

3. 从泛型类中继承

上例中的Farm<T>类以及本章前面介绍的其他几个类都继承自一个泛型类型。在Farm<T>中，这个类型是一个接口IEnumerable<T>。这里Farm<T>在T上提供的约束也会在IEnumerable<T>中使用的T上添加一个额外的约束。这可以用于限制未约束的类型，但需要遵循一些规则。

首先，如果某个类型所继承的基类型中受到了约束，该类型就不能“解除约束”。也就是说，类型T在所继承的基类型中使用时，该类型必须受到至少与基类型相同的约束。例如，下面的代码是正确的：

```
class SuperFarm<T> : Farm<T>
    where T : SuperCow {}
```

因为T在Farm<T>中被约束为Animal，把它约束为SuperCow，就是把T约束为这些值的一个子集，所以这是可行的。但是，以下代码不会编译：

```
class SuperFarm<T> : Farm<T>
    where T : struct

{}
```

可以肯定地讲，提供给SuperFarm<T>的类型T不能转换为可由Farm<T>使用的T，所以代码无法编译。

甚至对于约束为超集的情况，也会出现相同的问题：

```
class SuperFarm<T> : Farm<T>
    where T : class
{}

```

即使`SuperFarm<T>`允许存在像`Animal`这样的类型，`Farm<T>`中也不允许有满足类约束的其他类型。否则编译就会失败。这个规则适用于本章前面介绍的所有约束类型。

另外，如果继承自一个泛型类型，就必须提供所有必需的类型信息，这可以使用其他泛型类型参数的形式来提供，如上所述，也可以显式提供。这也适用于继承了泛型类型的非泛型类。例如：

```
public class Cards : List<Card>, ICloneable{}
```

这是可行的，但下面的代码会失败：

```
public class Cards : List<T>, ICloneable{}
```

因为没有提供`T`的信息，所以无法编译。

注意： 如果给泛型类型提供了参数，例如，上面的`List<Card>`，就可以称该类型是“关闭的”。同样，继承`List<T>`，就是继承一个“打开”的泛型类型。

4. 泛型运算符

在C#中，可以像其他方法一样进行运算符的重写，这也可以在泛型类中实现此类重写。例如，可在Farm<T>中定义如下隐式的转换运算符：

```
public static implicit operator List<Animal>(Farm<T> farm)
{
    List<Animal> result = new List<Animal>();
    foreach (T animal in farm)
    {
        result.Add(animal);
    }
    return result;
}
```

这样，如有必要，就可以在Farm<T>中把Animal对象直接作为List<Animal>来访问。例如，使用下面的运算符添加两个Farm<T>实例，这是很方便的：

```
public static Farm<T> operator +(Farm<T> farm1, List<T> farm2)
{
    Farm<T> result = new Farm<T>();
    foreach (T animal in farm1)
    {
        result.Animals.Add(animal);
    }
    foreach (T animal in farm2)
    {
        if (!result.Animals.Contains(animal))
```

```

    {
        result.Animals.Add(animal);
    }
}
return result;
}
public static Farm<T> operator +(List<T> farm1, Farm<T> farm2)
    => farm2 + farm1;

```

接着可以添加Farm<Animal>和Farm<Cow>的实例，如下所示：

```
Farm<Animal> newFarm = farm + dairyFarm;
```

在这行代码中，dairyFarm（是Farm<Cow>的实例）隐式转换为List<Animal>，List<Animal>可在Farm<T>中由重载运算符+使用。

读者可能认为，使用下面的代码也可以做到这一点：

```

public static Farm<T> operator +(Farm<T> farm1, Farm<T>

farm2){ ... }

```

但是，Farm<Cow>不能转换为Farm<Animal>，所以汇总会失败。为了更进一步，可以使用下面的转换运算符来解决这个问题：

```

public static implicit operator Farm<Animal>(Farm<T> farm)
{
    Farm<Animal> result = new Farm<Animal>();

```

```
foreach (T animal in farm)
{
    result.Animals.Add(animal);
}
return result;
}
```

使用这个运算符，`Farm<T>`的实例（如`Farm<Cow>`）就可以转换为`Farm<Animal>`的实例，这解决了上面的问题。所以，可以使用上面列出的两种方法中的一种，但是后者更适合，因为它比较简单。

5. 泛型结构

前几章说过，结构实际上与类相同，只有一些微小区别，而且结构是值类型，不是引用类型。所以，可以用与泛型类相同的方式来创建泛型结构。例如：

```
public struct MyStruct<T1, T2>
{
    public T1 item1;
    public T2 item2;
}
```

12.3.2 定义泛型接口

前面介绍了几个泛型接口，它们都位于`Systems.Collections.Generic`

名称空间中，例如，上一个示例中使用的IEnumerable<T>。定义泛型接口与定义泛型类所用的技术相同，例如：

```
interface MyFarmingInterface<T>
    where T : Animal
{
    bool AttemptToBreed(T animal1, T animal2);
    T OldestInHerd { get; }
}
```

其中，泛型参数T用作AttemptToBreed()的两个实参的类型和OldestInHerd属性的类型。

其继承规则与类相同。如果继承了一个基泛型接口，就必须遵循这些规则，例如保持基接口泛型类型参数的约束。

12.3.3 定义泛型方法

上个示例中使用了方法GetCows()。在讨论这个示例时也提到，可以使用泛型方法得到这个方法的更一般形式。本节将说明如何达到这一目标。在泛型方法中，返回类型和/或参数类型由泛型类型参数来确定。例如：

```
public T GetDefault<T>() => default(T);
```

这个小示例使用本章前面介绍的default关键字，为类型T返回默认值。这个方法的调用如下所示：


```
int myDefaultInt = GetDefault<int>();
```

在调用该方法时提供了类型参数T。

这个T与用于给类提供泛型类型参数的类型差异极大。实际上，可以通过非泛型类来实现泛型方法：

```
public class Defaulter
```

```
{
```

```
    public T GetDefault<T>() => default(T);
```

```
}
```

但如果类是泛型的，就必须为泛型方法类型使用不同的标识符。下面的代码不会编译：

```
public class Defaulter<T>
```

```
{
    public T GetDefault<T>() => default(T);
}
```

必须重命名方法或类使用的类型T。

泛型方法参数可以采用与类相同的方式使用约束，在此可以使用任意的类类型参数，例如：

```
public class Defaulter<T1>
```

```
{
    public T2
```

```
GetDefault<T2>
```

```
(
    where T2 : T1
```

```
{
    return default(T2
```

```
);  
    }  
}
```

其中，为方法提供的类型T2必须与给类提供的T1相同，或者继承自T1。这是约束泛型方法的常用方式。

在前面的Farm<T>类中，可以包含下面的方法（在Ch12Ex04的下载代码中包含它们，但已注释掉）。

```
public Farm<U> GetSpecies<U>() where U : T  
{  
    Farm<U> speciesFarm = new Farm<U>();  
    foreach (T animal in animals)  
    {  
        if (animal is U)  
        {  
            speciesFarm.Animals.Add(animal as U);  
        }  
    }  
    return speciesFarm;  
}
```

这可以替代GetCows()和相同类型的其他方法。这里使用的泛型类型参数U由T约束，T又由Farm<T>类约束为Animal。因此，如果愿意，可以把T的实例视为Animal的实例。

在Ch12Ex04的客户代码Program.cs中，使用这个新方法需要进行一处修改：

```
Farm<Cow> dairyFarm = farm.GetSpecies<Cow>
```

```
();
```

也可以编写如下代码：

```
Farm<Chicken> poultryFarm = farm.GetSpecies<Chicken>();
```

对于继承自Animal的其他类，都可以使用这种方法。

这里要注意，如果某个方法有泛型类型参数，会改变该方法的签名。也就是说，该方法有几个重载版本，它们仅在泛型类型参数上有区别。例如：

```
public void ProcessT<T
```

```
>(T op1){ ... }
```

```
public void ProcessT<T
```

```
, U
```

```
>(T op1){ ... }
```

使用哪个方法取决于调用方法时指定的泛型类型参数的个数。

12.3.4 定义泛型委托

最后一个要介绍的泛型类型是泛型委托。本章前面在介绍如何排序和搜索泛型列表时曾介绍过它们，即分别为此使用了`Comparison<T>`和`Predicate<T>`委托。

第6章介绍了如何使用方法的参数和返回类型、`delegate`关键字和委托名来定义委托，例如：

```
public delegate int MyDelegate(int op1, int op2);
```

要定义泛型委托，只需要声明和使用一个或多个泛型类型参数，例如：

```
public delegate T1 MyDelegate<T1, T2>(T2 op1, T2 op2) where T1
```

可以看出，也可以在这里使用约束。第13章将更详细地介绍委托，了解在常见的C#编程技术（即“事件”）中如何使用它们。

12.4 变体

变体（**variance**）是协变（**covariance**）和抗变（**contravariance**）的统称，这两个概念在.NET 4中引入。实际上，它们已经存在了较长时间了（在.NET 2.0中就可以使用），但在.NET 4之前很难实现它们，因为它们需要定制的编译过程。

要掌握这些术语的含义，最简单的方式是把它们与多态性进行比较。多态性允许把派生类型的对象放在基类型的变量中，例如：

```
Cow myCow = new Cow("Geronimo");  
Animal myAnimal = myCow;
```

其中把Cow类型的对象放在Animal类型的变量中，这是可行的，因为Cow派生自Animal。

但这不适用于接口，也就是说，下面的代码不能工作：

```
IMethaneProducer<Cow> cowMethaneProducer = myCow;  
IMethaneProducer<Animal> animalMethaneProducer = cowMethaneProc
```

假定Cow支持IMethaneProducer<Cow>接口，第一行代码就没有问题。但是，第二行代码预先假定两个接口类型有某种关系，但实际上这种关系不存在，所以无法把一种类型转换为另一种类型。是这样吗？使用本章前面介绍的技术肯定不行，因为泛型类型的所有类型参数都是不变的。但可以在泛型接口和泛型委托上定义变体类型参数，以适合上述代码演示的情形。

为使上述代码工作，IMethaneProducer<T>接口的类型参数T必须是协变的。有了协变的类型参数，就可以在IMethaneProducer<Cow>和IMethaneProducer<Animal>之间建立继承关系，这样一种类型的变量就可以包含另一种类型的值，这与多态性类似（但稍复杂些）。

为了完成对变体的介绍，需要看看变体的另一面：抗变。抗变和协变是类似的，但方向相反。抗变不能像协变那样，把泛型接口值放在使用基类型的变量中，但可以把该接口放在使用派生类型的变量中，例如：

```
IGrassMuncher<Cow> cowGrassMuncher = myCow;  
IGrassMuncher<SuperCow> superCowGrassMuncher = cowGrassMuncher
```

初看起来似乎有点古怪，因为不能通过多态性完成相同的功能。但是这在一些情况下是一项有效的技术，如“抗变”一节所述。

接下来的两节将介绍如何在泛型类型中实现变体，以及.NET Framework如何使用变体简化编程。

注意： 本节所有代码都包含在演示项目VarianceDemo中，可供使用。

12.4.1 协变

要把泛型类型参数定义为协变，可在类型定义中使用out关键字，

如下面的示例所示：

```
public interface IMethaneProducer<out T>{ ... }
```

对于接口定义，协变类型参数只能用作方法的返回值或属性`get`访问器。

说明协变用途的一个很好例子在.NET Framework中，即前面使用的`IEnumerable<T>`接口。在这个接口中，项类型`T`定义为协变，这表示可以把支持`IEnumerable<Cow>`的对象放在`IEnumerable<Animal>`类型的变量中。

因此下面的代码是有效的：

```
static void Main(string[] args)
{
    List<Cow> cows = new List<Cow>();
    cows.Add(new Cow("Geronimo"));
    cows.Add(new SuperCow("Tonto"));
    ListAnimals(cows);
    ReadKey();
}
static void ListAnimals(IEnumerable<Animal> animals)
{
    foreach (Animal animal in animals)
```



```

    {
        WriteLine(animal.ToString());
    }
}

```

其中cows变量的类型是List<Cow>，它支持IEnumerable<Cow>接口。通过协变，这个变量可以传送给需要IEnumerable<Animal>类型的参数的方法。回顾一下foreach循环的工作方式，就知道GetEnumerator()方法用于获取IEnumerator<T>的一个枚举器，该枚举器的Current属性用于访问项。IEnumerator<T>还将其类型参数定义为协变，这表示可以把它用作参数的get访问器，而且一切都运转良好。

12.4.2 抗变

要把泛型类型参数定义为抗变，可在类型定义中使用in关键字：

```

public interface IGrassMuncher<in T

>{ ... }

```

对于接口定义，抗变类型参数只能用作方法参数，不能用作返回类型。

理解这一点的最佳方式是列举一个在.NET Framework中使用抗变的例子。带有抗变类型参数的一个接口是前面用过的IComparer<T>。可以给Animal实现这个接口，如下所示：

```
public class AnimalNameLengthComparer : IComparer<Animal>
{
    public int Compare(Animal x, Animal y)
        => x.Name.Length.CompareTo(y.Name.Length);
}
```

这个比较器按名称的长度比较动物，所以可使用它对`List<Animal>`的实例排序。通过抗变，还可以使用它对`List<Cow>`的实例排序，尽管`List<Cow>.Sort()`方法需要`IComparer<Cow>`的实例。

```
List<Cow> cows = new List<Cow>();
cows.Add(new Cow("Geronimo"));
cows.Add(new SuperCow("Tonto"));
cows.Add(new Cow("Gerald"));
cows.Add(new Cow("Phil"));
cows.Sort(new AnimalNameLengthComparer());
```

大多数情况下，抗变都会发生——它被添加到.NET Framework中就是为了帮助执行这种操作。.NET 4及更高版本中这两种变体的优点是，可以在需要使用本节介绍的技术实现它。

12.5 练习

(1) 下面哪些元素可以是泛型？

a.类

b.方法

c.属性

d.运算符重载

e.结构

f.枚举

(2) 扩展Ch12Ex01中的Vector类，使*运算符返回两个矢量的点积（dot product）。

注意：两个矢量的点积定义为两个矢量的大小与两个矢量之间夹角余弦的乘积。

(3) 下面的代码存在什么错误？请加以修改。

```
public class Instantiator<T>
{
```

```

    public T instance;
    public Instantiator()
    {
        instance = new T();
    }
}

```

(4) 下面的代码存在什么错误？请加以修改。

```

public class StringGetter<T>
{
    public string GetString<T>(T item) => item.ToString();
}

```

(5) 创建一个泛型类ShortList<T>，它实现了IList<T>，包含一个项集合及集合的最大容量。这个最大容量应是一个整数，并可以提供给ShortList<T>的构造函数，或者默认为10。构造函数还应通过IEnumerable<T>参数获取项的最初列表。该类与List<T>的功能相同，但如果试图给集合添加太多的项，或者传递给构造函数的IEnumerable<T>包含太多的项，就会抛出IndexOutOfRangeException类型的异常。

(6) 下面的代码可以进行编译吗？试说明原因。

```

public interface IMethaneProducer<out T>
{
    void BelchAt(T target);
}

```

附录A给出了练习答案。

12.6 本章要点

主题	要点
使用泛型类型	泛型类型需要一个或多个类型参数才能工作。在声明变量时，传送需要的类型参数，就可以把泛型类型用作变量的类型。为此，应把逗号分隔的类型名列表放在尖括号中
可空类型	可空类型可使用指定值类型的任意值或null值。使用 <code>Nullable<T></code> 或 <code>T?</code> 语法，可以声明可空类型的变量
??运算符	空接合运算符返回第一个操作数的值，如果第一个操作数是null，就返回第二个操作数的值
泛型集合	泛型集合非常有用，因为它们内置了强类型化功能。可使用 <code>List<T></code> 、 <code>Collection<T></code> 和 <code>Dictionary<K, V></code> 等集合类型，它们还提供了泛型接口。为了针对泛型集合进行排序和搜索，应使用 <code>IComparer<T></code> 和 <code>IComparable<T></code> 接口
定义泛型类	泛型类型的定义十分类似于其他类型，但在指定类型名时需要添加泛型类型参数。与使用泛型类型一样，也需要把这些参数指定为逗号分隔的列表，并放在尖括号中。在使用类型名的地方都可以使用泛型类型参数，例如可在方法的返回值和参数中使用它们
泛型类型的参数约束	为了高效地在泛型类型代码中使用泛型类型参数，可以在使用类型时约束可以提供的类型。可以根据基类、所支持的接口、是否必须是值类型或引用类型以及是否支持无参数的构造函数等，来约束类型参数。如果没有这些约束，就必须使用 <code>default</code> 关键字来实例化泛型类型的变量
其他泛型类型	除类之外，还可以定义泛型接口、委托和方法

变体	<p>变体是类似于多态性的一个概念，但应用于类型参数。它允许使用一个泛型类型替代另一个泛型类型，这些泛型类型仅在所使用的泛型类型参数上有区别。协变允许在两种类型之间转换，其中目标类型有一个类型参数，它是源类型的类型参数的基类。抗变允许进行相反的转换。协变类型参数用out参数定义，只能用作返回类型和属性get访问器的类型。抗变类型参数用in参数定义，只能用作方法的参数</p>
----	--

第13章 高级C#技术

本章内容：

- ::运算符
- 全局名称空间限定符
- 如何创建定制异常
- 如何使用事件
- 如何使用匿名方法
- 如何使用C#特性
- 如何使用初始化器
- 使用var类型和类型推理
- 如何使用匿名类型
- 如何使用dynamic类型
- 如何使用命名和可选的方法参数
- 使用Lambda表达式

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 13 Code后，可找到与本章示例对应的单独文件。

本章将介绍前面未涉及的内容，继续讨论C#语言中不适合放在其他地方讨论的内容。C#的发明者Anders Hejlsberg和微软公司的其他人一直在更新和改进该语言。在撰写本书时，最新的改进都放在C#语言的第6版本中，它与.NET 4.6都作为Visual Studio 2015系列产品的一部分发布。阅读了本书前面的内容后，读者可能会考虑还需要什么其他功能。实际上，C#以前的版本从功能的角度来看并不缺乏什么，但这并不意味着无法进一步简化C#编程的某些方面，或者C#和其他技术之间的关系不能更加流畅。

本章还将对前面几章构建的CardLib代码进行最终的修改，并使用CardLib来创建扑克牌游戏。

13.1 ::运算符和全局名称空间限定符

::运算符提供了另一种访问名称空间中类型的方式。如果要使用一个名称空间的别名，但该别名与实际名称空间层次结构之间的界限不清晰，就必须使用::运算符。在那种情况下，名称空间层次结构优先于名称空间别名。为阐明其含义，考虑下列代码：

```
using MyNamespaceAlias = MyRootNamespace.MyNestedNamespace;
namespace MyRootNamespace
{
    namespace MyNamespaceAlias
    {
        public class MyClass {}
    }
    namespace MyNestedNamespace
    {
        public class MyClass {}
    }
}
```

MyRootNamespace中的代码使用以下代码引用一个类：

```
MyNamespaceAlias.MyClass
```

这行代码表示的类是

MyRootNamespace.MyNamespaceAlias.MyClass，而不是MyRootNamespace.MyNestedNamespace.MyClass。也就是说，MyRootNamespace.MyNamespaceAlias名称空间隐藏了由using语句定义的别名，该别名指向MyRootNamespace。MyNestedNamespace名称空间。仍然可以访问这个名称空间以及其中包含的类，但需要使用不同的语法：

```
MyNestedNamespace.MyClass
```

另外，还可以使用::运算符：

```
MyNamespaceAlias::MyClass
```

使用这个运算符会迫使编译器使用由using语句定义的别名，因此代码指向MyRootNamespace.MyNestedNamespace.MyClass。

::运算符还可以与global关键字一起使用，它实际上是顶级根名称空间的别名。这有助于更清晰地说明要指向哪个名称空间，如下所示：

```
global::System.Collections.Generic.List<int>
```

这是希望使用的类，即List<T>泛型集合类。它肯定不是用下列代码定义的类：

```
namespace MyRootNamespace
{
    namespace System
    {
        namespace Collections
        {
```

```
namespace Generic
{
    class List<T> {}
}
}
```

当然，应避免使名称空间的名称与已有的.NET名称空间相同，但这个问题只在大型项目中才会出现，作为大型开发队伍中的一员进行开发时，此类问题尤其严重。使用::运算符和global关键字可能是访问所需类型的唯一方式。

13.2 定制异常

第7章讨论了异常，以及如何使用try...catch...finally块处理它们。我们还论述了几个标准的.NET异常，包括异常的基类System.Exception。在应用程序中，有时也可以从这个基类中派生自己的异常类，并使用它们，而不是使用标准的异常。这样就可以把更具体的信息发送给捕获该异常的代码，让处理异常的捕获代码更有针对性。例如，可以给异常类添加一个新属性，以便访问某些底层信息，这样异常的接收代码就可以做出必要的改变，或者仅给出异常起因的更多信息。

注意：在System名称空间中两个基本的异常类ApplicationException和SystemException，它们派生于Exception。SystemException用作.NET Framework预定义的异常的基类，ApplicationException由开发人员用于派生自己的异常类。但最近的最佳做法是不从这个类中派生异常，而应使用Exception。

给CardLib添加定制异常

为演示定制异常的用法，最好通过升级CardLib项目来说明。目前，如果试图访问索引小于0或大于51的扑克牌，Deck.GetCard()方法目前就会抛出一个标准的.NET异常，但下面改为使用一个定制异常。

首先需要在BegVCSharp\Chapter13目录中创建一个新的类库项目

Ch13CardLib, 像以前一样把类从Ch12CardLib中复制过来, 并把名称空间改为Ch13CardLib。接着定义该异常。方法是使用在新类文件CardOutOfRangeException.cs中定义的一个新类, 这个新类是使用Project|Add Class添加到Ch13CardLib项目中的(这段代码包含在Ch13CardLib\CardOutOfRangeException.cs文件中):

```
public
```

```
class CardOutOfRangeException : Exception
```

```
{
```

```
    private Cards deckContents;
```

```
    public Cards DeckContents
```

```
{
```

```
    get { return deckContents; }
```

```
}
```

```
public CardOutOfRangeException(Cards sourceDeckContents)
```

```
: base("There are only 52 cards in the deck.")
```

```
{
```

```
    deckContents = sourceDeckContents;
```

```
}
```

```
}
```

这个类的构造函数需要使用Cards类的一个实例，它允许通过DeckContents属性来访问这个Cards对象，为Exception基类构造函数提供合适的错误信息，使该错误信息可以通过类的Message属性得到。

接着在Deck.cs中添加抛出该异常的代码，替换原来的标准异常（这段代码包含在Ch13CardLib\Deck.cs文件中）：

```
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum<= 51)
        return cards[cardNum];
    else
        throw new CardOutOfRangeException(cards.Clone()) as Car

}
```

DeckContents属性是通过对Deck对象的当前内容（其形式是一个Cards对象）进行深度复制来初始化的。这表示，此时的内容是异常抛出时的内容，所以随后对Deck内容的修改不会丢失这些信息。

要进行测试，使用下面的客户代码（这段代码包含在Ch13CardClient\Program.cs文件中）：

```
Deck deck1 = new Deck();
try
```

```
{  
    Card myCard = deck1.GetCard(60);  
}  
catch (CardOutOfRangeExcepction e)  
{  
    WriteLine(e.Message);  
    WriteLine(e.DeckContents[0]);  
}  
ReadKey();
```

结果如图13-1所示。

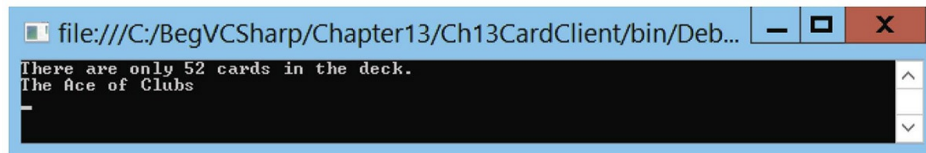


图13-1

其中捕获代码把异常的Message属性写到屏幕上。我们还通过DeckContents显示了Cards对象中的第一张牌，以证明可以通过定制的异常对象来访问Cards集合。

13.3 事件

本节主要讨论.NET中最常用的OOP技术：事件。像往常一样，首先介绍基础知识，分析事件到底是什么。之后讨论几个简单事件，看看使用它们可以做什么。然后论述如何创建和使用自己的事件。

本章最后介绍如何给CardLib类库添加一个事件，使该类库更完整。另外，因为这是在介绍一些更高级论题之前的最后一部分，我们还将创建一个使用该类库的有趣扑克牌游戏应用程序。

13.3.1 事件的含义

事件类似于异常，因为它们都由对象引发（抛出），并且都可以通过我们提供的代码来处理。但它们也有几个重要区别。最重要的区别是并没有与try...catch类似的结构来处理事件，你必须订阅（subscribe）它们。订阅一个事件的含义是提供代码，在事件发生时执行这些代码，它们称为事件处理程序。

单个事件可供多个处理程序订阅，在该事件发生时，这些处理程序都会被调用，其中包括引发该事件的对象所在的类中的事件处理程序，但事件处理程序也可能在其他类中。

事件处理程序本身都是简单方法。对事件处理方法的唯一限制是它必须匹配事件所要求的返回类型和参数。这个限制是事件定义的一部分，由一个委托指定。

注意： 在事件中使用委托是非常有用的。第6章介绍了委托，读者可以温习这一部分，复习一下委托是什么以及如何使用它们。

基本处理过程如下所示：首先，应用程序创建一个可以引发事件的对象。例如，假定一个即时消息传送（**instant messaging**）应用程序创建的对象表示一个远程用户的连接。当接收到远程用户通过该连接传送来的消息时，这个连接对象会引发一个事件，如图13-2所示。

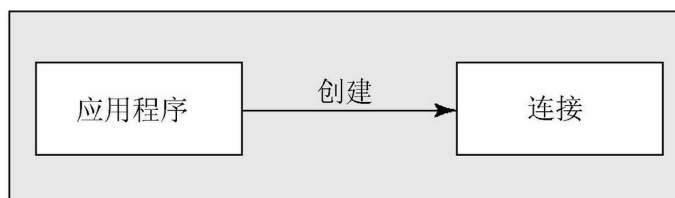


图13-2

接着，应用程序订阅事件。为此，即时消息传送应用程序将定义一个方法，该方法可以与事件指定的委托类型一起使用，把这个方法的一个引用传送给事件，而事件的处理方法可以是另一个对象的方法，例如当接收到消息时进行显示的显示设备对象，如图13-3所示。

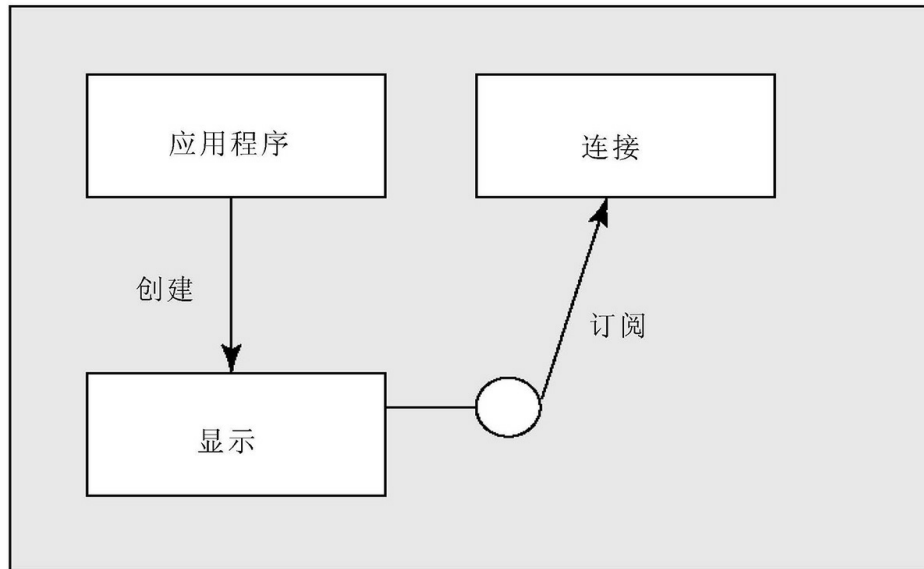


图13-3

引发事件后，就通知订阅器。当接收到通过连接对象传来的即时消息时，就调用显示设备对象上的事件处理方法。因为我们使用的是一个标准方法，所以引发事件的对象可以通过参数传送任何相关的信息，这样就大大增加了事件的通用性。在本例中，一个参数是即时消息的文本，事件处理程序可以在显示设备对象上显示它，如图13-4所示。

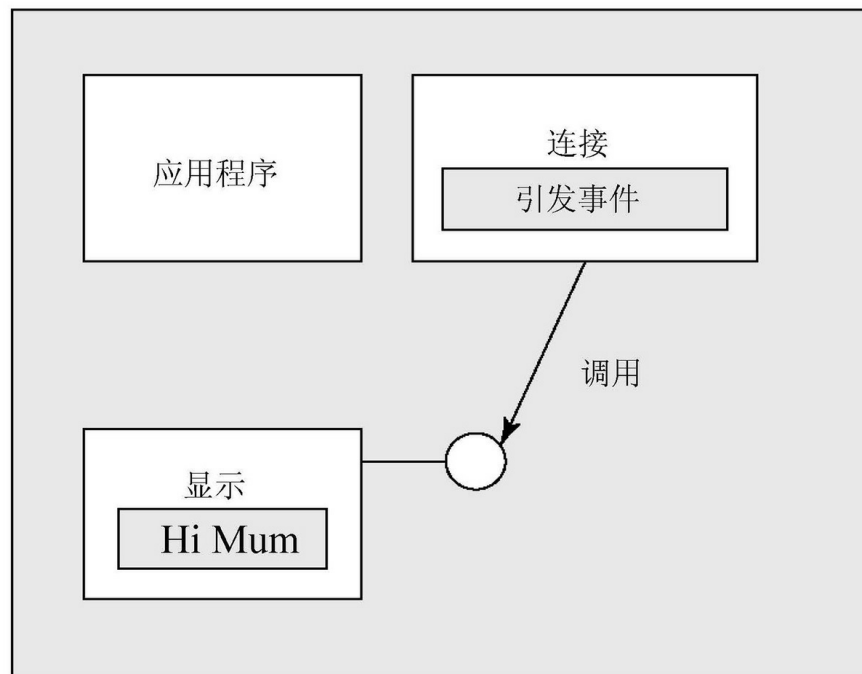


图13-4

13.3.2 处理事件

如前所述，要处理事件，需要提供一个事件处理方法来订阅事件，该方法的返回类型和参数应该匹配事件指定的委托。下面的示例使用一个简单的计时器对象引发事件，调用一个处理方法。

试一试：处理事件：**Ch13Ex01\Program.cs**

(1) 在C:\BegVCSharp\Chapter13目录中创建一个新的控制台应用程序Ch13Ex01。

(2) 修改Program.cs中的代码，如下所示：

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.Timers;
```

```
using static System.Console;
```

```
namespace Ch13Ex01  
{  
    class Program  
    {  
        static int counter = 0;  
  
  
        static string displayString =
```

"This string will appear one letter at a time

```
static void Main(string[] args)
```

```
{
```

```
    Timer myTimer = new Timer(100);
```

```
    myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

```
    myTimer.Start();
```

```
    System.Threading.Thread.Sleep(200);
```

```
    ReadKey();
```

```
}
```

```
static void WriteChar(object source, ElapsedEventArgs e)

{

    Write(displayString[counter++ % displayString.Length]);

}

}

}
```

(3) 运行应用程序（启动后，按任意键将终止执行程序），在经过短暂运行后，将显示如图13-5所示的结果。

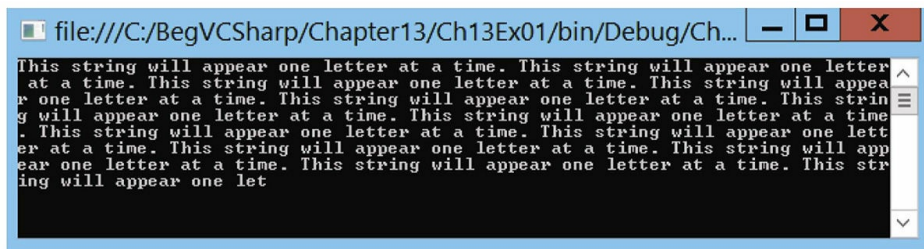


图13-5

示例的说明

用于引发事件的对象是System.Timers.Timer类的一个实例。使用一个时间段（以毫秒为单位）来初始化该对象。当使用Start()方法启动Timer对象时，就引发一系列事件，根据指定的时间段来引发事件。Main()用100毫秒初始化Timer对象，所以在启动该对象后，1秒钟内将引发10次事件：

```
static void Main(string[] args)
{
    Timer myTimer = new Timer(100);
```

Timer对象有一个Elapsed事件，这个事件要求事件处理程序必须匹配System.Timers.ElapsedEventHandler委托类型的返回类型和参数，该委托是.NET Framework中定义的标准委托之一，指定了如下所示的返回类型和参数：

```
void<MethodName>
>(object source, ElapsedEventArgs e);
```

Timer对象的第一个参数是它本身的引用，第二个参数则是ElapsedEventArgs对象的一个实例。现在可以不考虑这些参数，后面将论述它们。

在代码中，有一个匹配该返回类型和参数的方法：

```
static void WriteChar(object source, ElapsedEventArgs e)
```



```
{  
    Write(displayString[counter++ % displayString.Length]);  
}
```

这个方法使用Program的两个静态字段counter和displayString来显示一个字符。每次调用该方法时，显示的字符都不相同。

下一个任务是把这个处理程序与事件关联起来——即订阅它。为此，可以使用+=运算符，给事件添加一个处理程序，其形式是使用事件处理方法初始化的一个新委托实例：

```
static void Main(string[] args)  
{  
    Timer myTimer = new Timer(100);  
    myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

这个命令（使用有点古怪的语法，专用于委托）在列表中添加一个处理程序，当引发Elapsed事件时，就会调用该处理程序。可给这个列表添加任意多个处理程序，只要它们满足指定的条件即可。当引发事件时，会依次调用每个处理程序。

Main()剩余的任务是启动计时器：

```
myTimer.Start();
```

我们不想在处理完任何事件前终止应用程序，所以要让Main()函数一直执行。最简单的方式是请求用户输入，因为这个命令要在用户按下任意键后，才会停止处理。

```
ReadKey();
```

这里，Main()中的处理会停止，但Timer对象中的处理将继续。当该对象引发事件时，就调用WriteChar()方法，同时该方法运行Console.ReadLine()语句。使用System.Threading.Thread.Sleep（200）语句是为了让计时器有机会把消息发送给控制台应用程序。

注意，可使用上一章介绍的方法组概念来稍简化添加事件处理程序的语法：

```
myTimer.Elapsed += WriteChar  
  
;
```

最终结果是完全相同的，但不必显式指定委托类型，编译器会根据使用事件的上下文来指定它。但是，许多程序员不喜欢这个语法，因为它降低了可读性——不再能一眼看出使用了什么委托类型。如果喜欢，就可以使用这个语法。但为了清晰起见，本章使用的所有委托都显式指定。

13.3.3 定义事件

接着论述如何定义和使用自己的事件。我们将使用本节前面介绍的即时消息传送应用程序示例，并创建一个Connection对象，该对象引发由Display对象处理的事件。

试一试：定义事件： **Ch13Ex02**

(1) 在C:\BegVCSharp\Chapter13目录中创建一个新控制台应用程序Ch13Ex02。

(2) 添加一个新类Connection，并修改Connection.cs，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Timers;
```

```
using static System.Console;
```

```
namespace Ch13Ex02
{
    public delegate void MessageHandler(string messageText);
```

public

class Connection

{

public event MessageHandler MessageArrived;

private Timer pollTimer;

public Connection()

{

pollTimer = new Timer(100);

pollTimer.Elapsed += new ElapsedEventHandler(CheckForMe

}

public void Connect() => pollTimer.Start();

```
public void Disconnect() => pollTimer.Stop();
```

```
private static Random random = new Random();
```

```
private void CheckForMessage(object source, ElapsedEvent/
```

```
{
```

```
    WriteLine("Checking for new messages.");
```

```
    if ((random.Next(9) == 0) && (MessageArrived != null))
```

```
{
```

```
MessageArrived("Hello Mami!");
```

```
}
```

```
}
```

```
}
```

```
}
```

(3) 添加一个新类Display，并修改Display.cs，如下：

```
namespace Ch13Ex02
```

```
{
```

```
    public
```

```
class Display
```

```
{
```

```
public void DisplayMessage(string message)
```

```
=> WriteLine($"Message arrived: {message}");
```

```
}
```

```
}
```

(4) 修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)
```

```
{
```

```
    Connection myConnection = new Connection();
```

```
    Display myDisplay = new Display();
```

```
    myConnection.MessageArrived +=
```

```
        new MessageHandler (myDisplay.DisplayMessage);

myConnection.Connect();

ReadKey();

}
```

(5) 运行应用程序，其结果如图13-6所示。

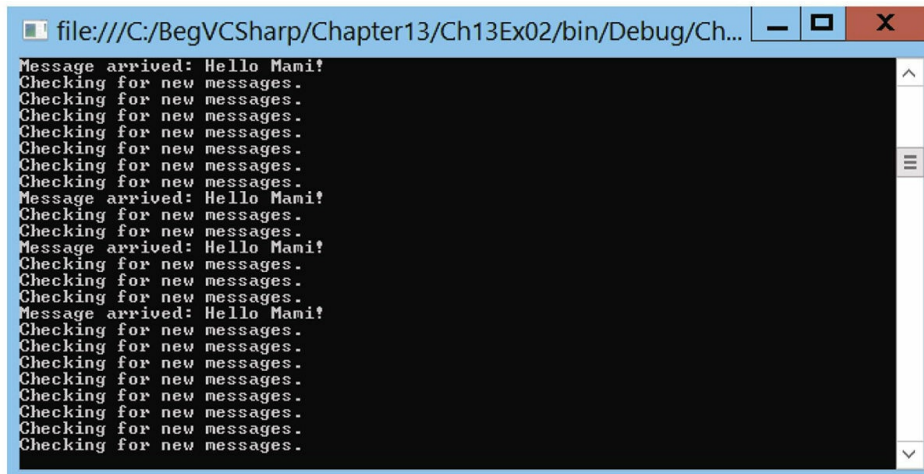


图13-6

示例的说明

这个应用程序中的大部分工作是由Connection类完成的。这个类的实例使用如本章第一个示例中所示的Timer对象，在类的构造函数中初始化它，并通过Connect()和Disconnect()访问它的状态（可访问和禁止访问）：

```
public class Connection
{
    private Timer pollTimer;
    public Connection()
    {
        pollTimer = new Timer(100);
        pollTimer.Elapsed += new ElapsedEventHandler(CheckForMess;
    }
    public void Connect() => pollTimer.Start();
    public void Disconnect() => pollTimer.Stop();
    ...
}
```

在构造函数中，我们还以与第一个示例相同的方式注册了Elapsed事件的一个事件处理程序。每当调用这个处理程序方法CheckForMessage()的次数达到10次后，就会引发一个事件。在分析它的代码前，首先来分析事件的定义。

在定义事件前，必须首先定义一个委托类型，以用于该事件，这个委托类型指定了事件处理方法必须拥有的返回类型和参数。为此，我们使用标准的委托语法，在Ch13Ex02名称空间中将该委托定义为公共类型，使该类型可供外部代码使用：

```
namespace Ch13Ex02
{
    public delegate void MessageHandler(string messageText);
```

这个委托类型称为**MessageHandler**，是**void**方法的签名，它有一个**string**参数。使用这个参数可以把**Connection**对象收到的即时消息发送给**Display**对象。定义了委托（或者找到合适的现有委托）后，就可以把事件本身定义为**Connection**类的一个成员：

```
public class Connection
{
    public event MessageHandler MessageArrived;
```

给事件命名（这里使用名称**MessageArrived**），在声明时，使用**event**关键字，并指定要使用的委托类型（前面定义的**MessageHandler**委托类型）。以这种方式声明事件后，就可以引发它，做法是按名称来调用它，就像它是一个其返回类型和参数是由委托指定的方法一样。例如，使用下面的代码引发这个事件：

```
MessageArrived("This is a message.");
```

如果定义该委托时不包含任何参数，就可以使用下面的代码：

```
MessageArrived();
```

如果定义了较多参数，就需要用比较多的代码来引发事件。**CheckForMessage()**方法如下所示：

```
private static Random random = new Random();
private void CheckForMessage(object source, ElapsedEventArgs
```

```

{
    WriteLine("Checking for new messages.");
    if ((random.Next(9) == 0) && (MessageArrived != null))
    {
        MessageArrived("Hello Mami!");
    }
}

```

使用前面几章中的Random类实例，生成一个介于0~9之间的随机数，如果该随机数为0，就引发一个事件，它的发生几率为10%。这类类似于轮询连接，看看是否接收到消息，不可能每次检测时，都没有接收到消息。为将计时器与Connection的实例分隔开，使用了Random类的一个私有静态实例。

注意，这里还提供了其他逻辑。只有表达式MessageArrived !=null为true，才引发一个事件。这个表达式也使用了委托语法，但语法稍有不同，其含义是“事件是否有订阅者？”。如果没有订阅者，MessageArrived就是null，也就不会引发事件。

订阅事件的类是Display，它包含一个方法DisplayMessage()，其定义如下所示：

```

public class Display
{
    public void DisplayMessage(string message)
        => WriteLine($"Message arrived: {message}");
}

```

这个方法匹配委托类型（而且是公共的，如果类不是生成该事件的类，则其事件处理程序必须是公共的），所以可用它来响应 `MessageArrived` 事件。

剩下的是 `Main()` 中的代码初始化了 `Connection` 和 `Display` 类的实例，把它们关联起来，开始执行任务。这里需要的代码类似于第一个示例中的代码：

```
static void Main(string[] args)
{
    Connection myConnection = new Connection();
    Display myDisplay = new Display();
    myConnection.MessageArrived +=
        new MessageHandler(myDisplay.DisplayMessage);
    myConnection.Connect();
    System.Threading.Thread.Sleep(200);
    ReadKey();
}
```

再次调用 `ReadKey()`，当开始执行 `Connection` 对象的 `Connect()` 方法并增加一段延迟时间后，暂停 `Main()` 的处理。

1. 多用途的事件处理程序

前面 `Timer.Elapsed` 事件的委托包含了事件处理程序中常见的两类参数，如下所示：

- `object source`——引发事件的对象的引用
- `ElapsedEventArgs e`——由事件传送的参数

在这个事件（以及许多其他的事件）中使用`object`类型参数的原因是，我们常常要为由不同对象引发的几个相同事件使用同一个事件处理程序，但仍要指定哪个对象生成了事件。

要说明这一点，下面将扩展上一个示例。

试一试：使用多用途的事件处理程序：**Ch13Ex03**

（1）在C:\BegVCSharp\Chapter13目录中创建一个新控制台应用程序Ch13Ex03。

（2）复制Ch13Ex02中Program.cs、Connection.cs和Display.cs的代码，并将每个文件中的Ch13Ex02名称空间改成Ch13Ex03。

（3）添加一个新类MessageArrivedEventArgs，修改MessageArrivedEventArgs.cs，如下所示：

```
namespace Ch13Ex03
{
    public

class MessageArrivedEventArgs : EventArgs
```

```
{  
    private string message;  
  
    public string Message  
  
    {  
  
        get { return message; }  
  
    }  
  
    public MessageArrivedEventArgs()
```

```
{
```

```
    message = "No message sent.";
```

```
}
```

```
public MessageArrivedEventArgs(string newMessage)
```

```
{
```

```
    message = newMessage;
```

```
}
```

```
}  
}
```

(4) 修改Connection.cs, 如下所示:

```
namespace Ch13Ex03  
{  
    // delegate definition removed  
  
    public class Connection  
    {  
        public event EventHandler<MessageArrivedEventArgs>  
  
        MessageArrived;  
  
        public string Name { get; set; }  
  
        ...  
  
        private void CheckForMessage(object source, EventArgs e)
```



```

    {
        WriteLine("Checking for new messages.");
        if ((random.Next(9) == 0) && (MessageArrived != null))
        {
            MessageArrived(this, new MessageArrivedEventArgs(

"Hello Mami!")

);
        }
    }
    ...
}
}

```

(5) 修改Display.cs, 如下所示:

```

    public void DisplayMessage(object source, MessageArrivedE\

)

    {
        WriteLine($"Message arrived from: {((Connection)source)

```

```
        WriteLine($"Message Text: {e.Message}");
    }
}
```

(6) 修改Program.cs, 如下所示:

```
static void Main(string[] args)
{
    Connection myConnection1 = new Connection();

    myConnection1.Name = "First connection.";

    Connection myConnection2 = new Connection();

    myConnection2.Name = "Second connection.";
```

```
Display myDisplay = new Display();
```

```
myConnection1.MessageArrived += myDisplay.DisplayMessag
```

```
myConnection2.MessageArrived += myDisplay.DisplayMessag
```

```
myConnection1.Connect();
```

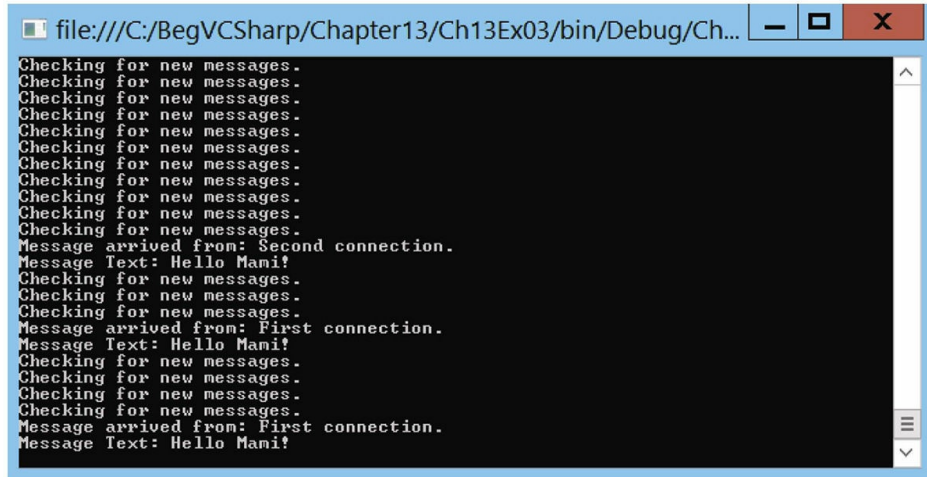
```
myConnection2.Connect();
```

```
System.Threading.Thread.Sleep(200);
```

```
ReadKey();
```

```
}
```

(7) 运行应用程序，其结果如图13-7所示。



```
file:///C:/BegVCSharp/Chapter13/Ch13Ex03/bin/Debug/Ch...
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Message arrived from: Second connection.
Message Text: Hello Mani!
Checking for new messages.
Checking for new messages.
Checking for new messages.
Message arrived from: First connection.
Message Text: Hello Mani!
Checking for new messages.
Checking for new messages.
Checking for new messages.
Checking for new messages.
Message arrived from: First connection.
Message Text: Hello Mani!
```

图13-7

示例的说明

发送一个引发事件的对象引用，将其作为事件处理程序的一个参数，就可以为不同对象定制处理程序的响应。利用该引用可以访问源对象，包括它的属性。

通过发送包含在派生于System.EventArgs（与ElapsedEventArgs相同）的类中的参数，就可以将其他必要信息提供为参数（例如，MessageArrivedEventArgs类上的Message参数）。

另外，这些参数也将得益于多态性。可为MessageArrived事件定义一个处理程序，如下所示：

```
public void DisplayMessage(object source, EventArgs e

)

{
```

```
WriteLine($"Message arrived from: {((Connection)source)}.  
WriteLine($"Message Text: {((MessageArrivedEventArgs)e).  
}
```

这个应用程序将像以前那样执行，但DisplayMessage()方法变得更加通用（至少从理论上讲是这样的——需要使用更多实现代码，才能满足生产环境的要求）。这个处理程序还可以处理其他事件，例如Timer.Elapsed事件，但必须修改处理程序的内部代码，这样，在引发这个事件时，发送过来的参数才会得到正确处理（以这种方式把它们转换为Connection和MessageArrivedEventArgs对象，会抛出一个异常，所以这里应使用as运算符，检查null值）。

2. EventHandler和泛型EventHandler<T>类型

大多数情况下，都应遵循上一节提出的模式，使用返回类型为void、带两个参数的事件处理程序。第一个参数的类型是object，是事件源。第二个参数的类型派生于System.EventArgs，包含任意事件实参。这非常常见，为此.NET提供了两个委托类型EventHandler和EventHandler<T>，以便定义事件。它们都是委托，使用标准的事件处理模式。泛型版本允许指定要使用的事件实参的类型。

在前面的示例中演示了这一点，使用了泛型委托类型EventHandler<T>，如下所示：

```
public class Connection
```

```
{  
    public event EventHandler<MessageArrivedEventArgs>  
  
    MessageArrived;  
  
    ...  
}
```

这显然是件好事，因为它简化了代码。一般来说，在定义事件时，最好使用这些委托类型。注意，如果事件不需要事件实参数据，仍然可以使用EventHandler委托类型，只不过要传递EventArgs.Empty作为实参值。

3. 返回值和事件处理程序

前面的所有事件处理程序都使用void类型的返回值。可以为事件提供返回类型，但这会出问题。这是因为引发给定的事件，可能会调用多个事件处理程序。如果这些处理程序都返回一个值，那么我们不知道该使用哪个返回值。

系统处理这个问题的方式是，只允许访问由事件处理程序最后返回的那个值，也就是最后一个订阅该事件的处理程序返回的值。这个功能在某些情况下是有用的，但最好使用void类型的事件处理程序，且避免使用out类型的参数（如果使用out参数，参数返回的值的源头就是不清楚的）。

4. 匿名方法

除了定义事件处理方法外，还可以选择使用匿名方法（anonymous method）。匿名方法实际上并非传统意义上的方法，它不是某个类上的方法，而纯粹是为用作委托目的而创建的。

要创建匿名方法，需要使用下面的代码：

```
delegate(parameters)
```

```
{  
    // Anonymous method code.  
};
```

其中parameters是一个参数列表，这些参数匹配正在实例化的委托类型，由匿名方法的代码使用，例如：

```
delegate(Connection source, MessageArrivedEventArgs e)
```

```
{  
    // Anonymous method code matching MessageHandler event in C  
};
```

例如，使用这段代码可以完全绕过Ch13Ex03中的Display.DisplayMessage()方法：

```
myConnection1.MessageArrived +=
```

```
delegate(Connection source, MessageArrivedEventArgs  
  
{  
  
    WriteLine($"Message arrived from: {source.Name}");  
  
    WriteLine($"Message Text: {e.Message}");  
  
};
```

使用匿名方法时要注意，对于包含它们的代码块来说，它们是局部的，可以访问这个作用域内的局部变量。如果使用这样一个变量，它就成为外部变量（outer variable）。外部变量在超出作用域时，是不会删除的，这与其他局部变量不同，在使用它们的匿名方法被销毁时，才会删除外部变量。这比我们希望的时间晚一些，所以要格外小心。如果外部变量占用了大量内存，或者使用的资源在其他方面是比较昂贵的（例

如资源数量有限），就可能导致内存或性能问题。

13.4 扩展和使用CardLib

前面介绍了事件的定义和使用，现在就可以在Ch13CardLib中使用它们了。在库中需要添加一个LastCardDrawn事件，当使用GetCard获得Deck对象中的最后一个Card对象时，就将引发该事件。这个事件允许订阅者（subscriber）自动重新洗牌，减少需要在客户端完成的处理。这个事件将使用EventHandler委托类型，并传递一个Deck对象的引用作为事件源，这样无论处理程序在什么地方，都可以访问Shuffle()方法。在Deck.cs中添加以下代码以定义并引发事件（这段代码包含在Ch13CardLib\Deck.cs文件中）：

```
namespace Ch13CardLib
{
    public class Deck : ICloneable

    {

        public event EventHandler LastCardDrawn;
```

```

...
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum<= 51)
    {

        if ((cardNum == 51) && (LastCardDrawn != null))

            LastCardDrawn(this, EventArgs.Empty);

        return cards[cardNum];
    }

    else
        throw new CardOutOfRangeException((Cards)cards.Clone());
}
...
}

```

这是把事件添加到Deck类定义需要的所有代码。

开发CardLib库后，就可以使用它了。在结束讲述C#和.NET Framework中OOP技术的这个部分前，我们将编写扑克牌应用程序的基本代码，其中将使用我们熟悉的扑克牌类。

与前面的章节一样，我们将在Ch13CardLib解决方案中添加一个客户控制台应用程序，添加一个Ch13CardLib项目的引用，使其成为启动项目。这个应用程序称为Ch13CardClient。

首先在Ch13CardClient的一个新文件Player.cs中创建一个新类Player，相应代码可在本章下载代码的Ch13CardClient\Player.cs文件中找到。这个类包含两个自动属性：Name（字符串）和PlayHand（Cards类型）。这些属性有私有的set访问器。但是PlayHand属性仍可以对其内容进行写入访问，这样就可以修改玩家手中的扑克牌。

我们还把默认的构造函数设置为私有，以隐藏它，并提供了一个公共的非默认构造函数，该函数接受Player实例中Name属性的初始值。

最后提供一个bool类型的方法HasWon()。如果玩家手中的扑克牌花色都相同（一个简单的取胜条件，但并没有什么意义），该方法就返回true。

Player.cs的代码如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;  
using Ch13CardLib;
```

```
namespace Ch13CardClient  
{  
    public
```

```
class Player  
{  
    public string Name { get; private set; }  
  
    public Cards PlayHand { get; private set; }  
  
    private Player()  
  
    {
```

```
}
```

```
public Player(string name)
```

```
{
```

```
    Name = name;
```

```
    PlayHand = new Cards();
```

```
}
```

```
public bool HasWon()
```

```
{
```

```
    bool won = true;
```

```
    Suit match = PlayHand[0].suit;
```

```
    for (int i = 1; i<PlayHand.Count; i++)
```

```
    {
```

```
        won &= PlayHand[i].suit == match;
```

```
    }
```

```
        return won;

    }

}

}
```

接着定义一个处理扑克牌游戏的类Game，这个类在Ch13CardClient项目的Game.cs文件中。这个类有4个私有成员字段：

- playDeck——Deck类型的变量，包含要使用的一副扑克牌
- currentCard——一个int值，用作下一张要翻开的扑克牌的指针
- players——一个Player对象数组，表示游戏玩家
- discardedCards——Cards集合，表示玩家扔掉的扑克牌，但还没有放回整副牌中。

这个类的默认构造函数初始化了存储在playDeck中的Deck，并洗牌，把currentCard指针变量设置为0（playDeck中的第一张牌），并关联了playDeck.LastCardDrawn事件的处理程序Reshuffle()。这个处理程序将洗牌，初始化discardedCards集合，并将currentCard重置为0，准备从新的一副牌中读取扑克牌。

Game类还包含两个实用方法：SetPlayers()可以设置游戏的玩家（Player对象数组），DealHands()给玩家发牌（每个玩家有7张牌）。玩家的数量限制为2~7人，确保每个玩家有足够多的牌。

最后，PlayGame()方法包含游戏逻辑。我们将在分析了Program.cs中的代码后介绍这个方法，Game.cs的剩余代码如下所示（这段代码包含在Ch13CardClient\Game.cs文件中）：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Ch13CardLib;
```

```
using static System.Console;
```

```
namespace Ch13CardClient
{
    public

class Game
```

```
{  
    private int currentCard;  
  
    private Deck playDeck;  
  
    private Player[] players;  
  
    private Cards discardedCards;  
  
    public Game()  
  
    {  
  
        currentCard = 0;  

```

```
playDeck = new Deck(true);
```

```
playDeck.LastCardDrawn += Reshuffle;
```

```
playDeck.Shuffle();
```

```
discardedCards = new Cards();
```

```
}
```

```
private void Reshuffle(object source, EventArgs args)
```

```
{
```

```
WriteLine("Discarded cards reshuffled into deck.");
```

```
((Deck)source).Shuffle();
```

```
discardedCards.Clear();
```

```
currentCard = 0;
```

```
}
```

```
public void SetPlayers(Player[] newPlayers)
```

```
{
```

```
    if (newPlayers.Length > 7)
```

```
    {
        throw new ArgumentException(
```

```
            "A maximum of 7 players may play this game.");
```

```
    }
    if (newPlayers.Length < 2)
```

```
    {
        throw new ArgumentException(
```

```
            "A minimum of 2 players may play this game.");
```

```
players = newPlayers;
```

```
}
```

```
private void DealHands()
```

```
{
```

```
for (int p = 0; p<players.Length; p++)
```

```
{
```

```
for (int c = 0; c<7; c++)
```

```
{
```

```
    players[p].PlayHand.Add(playDeck.GetCard(currentCard));
```

```
}
```

```
}
```

```
}
```

```
public int PlayGame()
```

```
{
```

```
        // Code to follow.

    }

}

}
```

Program.cs中包含Main()方法，它初始化并运行游戏。这个方法执行以下步骤：

- (1) 显示引导画面。
- (2) 提示用户输入玩家数（2~7）。
- (3) 根据玩家数建立一个Player对象数组。
- (4) 给每个玩家取名，用于初始化数组中的一个Player对象。
- (5) 创建一个Game对象，使用SetPlayers()方法指定玩家。
- (6) 使用PlayGame()方法启动游戏。
- (7) PlayGame()的int返回值用于显示一条获胜消息（返回的值是

Player对象数组中获胜的玩家的索引）。

这个方法的代码（为清晰起见，加了一些注释）如下所示（这段代码包含在Ch13CardClient\Program.cs文件中）：

```
static void Main(string[] args)
{
    // Display introduction.

    WriteLine("BenjaminCards: a new and exciting card game.

    WriteLine("To win you must have 7 cards of the same sui

    " your hand.");

    WriteLine();
```

```
// Prompt for number of players.
```

```
bool inputOK = false;
```

```
int choice = -1;
```

```
do
```

```
{
```

```
    WriteLine("How many players (2-7)?");
```

```
    string input = ReadLine();
```

```
try
```

```
{
```

```
    // Attempt to convert input into a valid number of |
```

```
    choice = Convert.ToInt32(input);
```

```
    if ((choice >= 2) && (choice<= 7))
```

```
        inputOK = true;
```

```
}
```

```
catch
```

```
{
```

```
    // Ignore failed conversions, just continue prompt
```

```
}
```

```
} while (inputOK == false);
```

```
// Initialize array of Player objects.
```

```
Player[] players = new Player[choice];
```

```
// Get player names.
```

```
for (int p = 0; p<players.Length; p++)
```

```
{
```

```
    WriteLine($"Player {p + 1}, enter your name:");
```

```
    string playerName = ReadLine();
```

```
    players[p] = new Player(playerName);
```

```
}
```

```
// Start game.
```

```
Game newGame = new Game();
```

```
newGame.SetPlayers(players);
```

```
int whoWon = newGame.PlayGame();
```

```
// Display winning player.
```

```
WriteLine($"{players[whoWon].Name} has won the game!");
```

```
ReadKey();
```

```
}
```

接着分析一下应用程序的主体**PlayGame()**。由于篇幅所限，这里不准备详细讲解这个方法，而只是加注了一些注释，使其更容易理解。实际上，这些代码都不复杂，仅是较多而已。

每个玩家都可以查看手中的牌和桌面上的一张翻开的牌。他们可以拾取这张牌，或者翻开一张新牌。在拾取一张牌后，玩家必须扔掉一张牌，如果他们拾取了桌面上的那张牌，就必须用另一张牌替换桌面上的那张牌，或者把扔掉的那张牌放在桌面上那张牌的上面（把扔掉的那张牌添加到**discardedCards**集合中）。

在分析这段代码时，一个关键问题在于**Card**对象的处理方式。必须清楚，这些对象定义为引用类型，而不是值类型（使用结构）。给定的**Card**对象似乎同时存在于多个地方，因为引用可以存在于**Deck**对象、**Player**对象的**hand**字段、**discardedCards**集合和**playCard**对象（桌面上的当前牌）中。这样便于跟踪扑克牌，特别是可以用于从一副牌中拾取一张新牌。如果牌不在任何玩家的手中，也不在**discardedCards**集合中，才能接受该牌。

代码如下所示：

```
public int PlayGame()
```

```
{  
    // Only play if players exist.  
  
    if (players == null)  
  
        return -1;  
  
    // Deal initial hands.  
  
    DealHands();  
  
    // Initialize game vars, including an initial card to plac  
  
    // table: playCard.
```



```
bool GameWon = false;
```

```
int currentPlayer;
```

```
Card playCard = playDeck.GetCard(currentCard++);
```

```
discardedCards.Add(playCard);
```

```
// Main game loop, continues until GameWon == true.
```

```
do
```

```
{
```

```
// Loop through players in each game round.
```

```
for (currentPlayer = 0; currentPlayer < players.Length;
```

```
    currentPlayer++)
```

```
{
```

```
    //Write out current player, player hand, and the card c
```

```
    // table.
```

```
WriteLine($"{players[currentPlayer].Name}'s turn.");
```

```
WriteLine("Current hand:");
```

```
foreach (Card card in players[currentPlayer].PlayHand)
```

```
{
```

```
    WriteLine(card);
```

```
}
```

```
WriteLine($"Card in play: {playCard}");
```

```
// Prompt player to pick up card on table or draw a new c
```

```
bool inputOK = false;
```

```
do
```

```
{
```

```
    WriteLine("Press T to take card in play or D to draw:")
```

```
    string input = ReadLine();
```

```
    if (input.ToLower() == "t")
```

```
{
```

```
    // Add card from table to player hand.
```

```
    WriteLine("Drawn: {playCard}");
```

```
    // Remove from discarded cards if possible (if deck
```

```
    // is reshuffled it won't be there any more)
```

```
    if (discardedCards.Contains(playCard))
```

```
{
```

```
discardedCards.Remove(playCard);
```

```
}
```

```
players[currentPlayer].PlayHand.Add(playCard);
```

```
inputOK = true;
```

```
}
```

```
if (input.ToLower() == "d")
```

```
{
```

```
// Add new card from deck to player hand.
```

```
Card newCard;
```

```
// Only add card if it isn't already in a player han
```

```
// or in the discard pile
```

```
bool cardIsAvailable;
```

```
do
```

```
{
```

```
newCard = playDeck.GetCard(currentCard++);
```

```
// Check if card is in discard pile
```

```
cardIsAvailable = !discardedCards.Contains(newCard,
```

```
if (cardIsAvailable)
```

```
{
```

```
// Loop through all player hands to see if newCar
```



```
// is already in a hand.
```

```
foreach (Player testPlayer in players)
```

```
{
```

```
    if (testPlayer.PlayHand.Contains(newCard))
```

```
    {
```

```
        cardIsAvailable = false;
```

```
        break;
```

```
}
```

```
}
```

```
}
```

```
} while (!cardIsAvailable);
```

```
// Add the card found to player hand.
```

```
WriteLine($"Drawn: {newCard}");
```

```
players[currentPlayer].PlayHand.Add(newCard);
```

```
inputOK = true;
```

```
}
```

```
} while (inputOK == false);
```

```
// Display new hand with cards numbered.
```

```
WriteLine("New hand:");
```

```
for (int i = 0; i<players[currentPlayer].PlayHand.Count;
```

```
{
```

```
    WriteLine($"{i + 1}: " +
```

```
        $"{ players[currentPlayer].PlayHand[i]}");
```

```
}
```

```
// Prompt player for a card to discard.
```

```
inputOK = false;
```

```
int choice = -1;
```

do

{

WriteLine("Choose card to discard:");

string input = ReadLine();

try

{

// Attempt to convert input into a valid card number

```
choice = Convert.ToInt32(input);
```

```
if ((choice > 0) && (choice<= 8))
```

```
inputOK = true;
```

```
}
```

```
catch
```

```
{
```

```
        // Ignore failed conversions, just continue prompting  
  
    }  
  
} while (inputOK == false);  
  
// Place reference to removed card in playCard (place the  
  
// on the table), then remove card from player hand and add  
  
// to discarded card pile.  
  
playCard = players[currentPlayer].PlayHand[choice - 1];
```

```
players[currentPlayer].PlayHand.RemoveAt(choice - 1);
```

```
discardedCards.Add(playCard);
```

```
WriteLine($"»Discarding: {playCard}»");
```

```
// Space out text for players
```

```
WriteLine();
```

```
// Check to see if player has won the game, and exit the
```

```
// loop if so.
```



```
GameWon = players[currentPlayer].HasWon();
```

```
if (GameWon == true)
```

```
    break;
```

```
}
```

```
} while (GameWon == false);
```

```
// End game, noting the winning player.
```

```
return currentPlayer;
```

}

图13-8显示了一个正在进行的游戏。



```
file:///C:/BegVCSharp/Chapter13/Ch13CardClient/bin/Deb...
BenjaminsCards: a new and exciting card game.
To win you must have 7 cards of the same suit in your hand.
How many players (2-7)?
2
Player 1, enter your name:
Ben
Player 2, enter your name:
Todd
Ben's turn.
Current hand:
The Ten of Hearts
The Six of Clubs
The King of Hearts
The Ten of Spades
The Eight of Hearts
The Seven of Clubs
The King of Spades
Card in play: The Nine of Hearts
Press I to take card in play or D to draw:
I
Drawn: The Nine of Hearts
New hand:
1: The Ten of Hearts
2: The Six of Clubs
3: The King of Hearts
4: The Ten of Spades
5: The Eight of Hearts
6: The Seven of Clubs
7: The King of Spades
8: The Nine of Hearts
Choose card to discard:
1
Discarding: The Ten of Hearts
Todd's turn.
Current hand:
The Jack of Spades
The Ten of Clubs
The Four of Spades
The Seven of Diamonds
The Four of Diamonds
The Three of Hearts
The Jack of Hearts
Card in play: The Ten of Hearts
Press I to take card in play or D to draw:
D
Drawn: The Six of Hearts
New hand:
1: The Jack of Spades
2: The Ten of Clubs
3: The Four of Spades
4: The Seven of Diamonds
5: The Four of Diamonds
6: The Three of Hearts
7: The Jack of Hearts
8: The Six of Hearts
Choose card to discard:
-
```

图13-8

作为最终的练习，仔细查看`Player.HasWon()`中的代码。有什么方法可以使这段代

13.5 特性

本节将简要介绍一种为使用所创建类型的代码提供额外信息的方法：特性（`attribute`）。

比如，我们要创建的某个类包含了一个极简单的方法。换句话说，这个方法简单到我

[DebuggerStepThrough]

```
public void DullMethod() { ... }
```

上述代码中所包含的特性就是**[DebuggerStepThrough]**。所有特性的添加方式都是

上述代码中所使用的特性实际上是通过**DebuggerStepThroughAttribute**这个类：

通过上述方式添加特性后，编译器就会创建该特性类的一个实例，然后将其与类方法

```
[DoesInterestingThings(1000, WhatDoesItDo = "voodoo")]
```

```
public class DecoratedClass {}
```

上述特性就将值1000传递给了DoesInterestingThingsAttribute的构造函数，

13.5.1 读取特性

要读取特性的值，我们必须使用一种称为“反射（`reflection`）”的技术。这种非常

简单来说，反射可以取得保存在Type对象（本书中会多次提到该对象）中的使用信息

为此，最简单的方法也就是本书将要为大家介绍的唯一方法，即通过Type.GetCust

例如，下面的代码可以列出DecoratedClass这个类的特性：

```
Type classType = typeof(DecoratedClass);
```

```

object[] customAttributes = classType.GetCustomAttributes(true)
foreach (object customAttribute in customAttributes)
{
    WriteLine($"Attribute of type {customAttribute} found.");
}

```

通过这种方法了解到不同的特性后，我们就可以为不同的特性采取不同的操作了。这

13.5.2 创建特性

只要通过`System.Attribute`类进行派生，我们也可以创建出自己的特性。一般来

另外，还需要为自己的特性做两个选择：要将其应用到什么类型的目标（类、属性或

例如，下面的代码指定了一个特性可以应用到类或属性中（一次）：

```

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Method,
                AllowMultiple = false)]
class DoesInterestingThingsAttribute : Attribute
{

```

```

public DoesInterestingThingsAttribute(int howManyTimes)
{
    HowManyTimes = howManyTimes;
}

public string WhatDoesItDo { get; set; }
public int HowManyTimes { get; private set; }
}

```

这样，就可以像前面的代码片段中看到的那样来使用**DoesInterestingThings**特性：

```

[DoesInterestingThings(1000, WhatDoesItDo = "voodoo")]
public class DecoratedClass {}

```

只要像下面这样修改前面的代码，就可以访问这一特性的属性：

```

Type classType = typeof(DecoratedClass);
object[] customAttributes = classType.GetCustomAttributes(true)
foreach (object customAttribute in customAttributes)
{
    WriteLine($"Attribute of type {customAttribute} found.");
    DoesInterestingThingsAttribute interestingAttribute =

```

```
customAttribute as DoesInterestingThingsAttribute;
```

```
if (interestingAttribute != null)
```

```
{
```

```
WriteLine($"This class does {interestingAttribute.WhatDoes
```

```
$" {interestingAttribute.HowManyTimes}!");
```

```
}
```

```
}
```


运用了本节讲到的各种方法后，最终代码将可以得到如图13-9所示的结果。

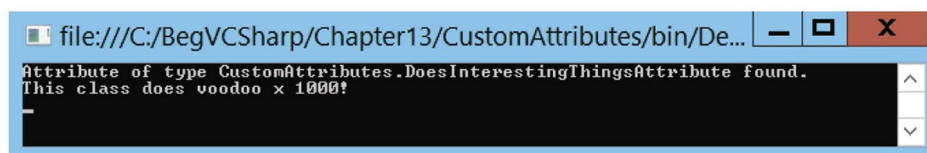


图13-9

“特性”这一技术在所有 .NET 应用程序都可以变得非常有用，特别是WPF和Windows

13.6 初始化器

前面的章节学习了如何用各种方式实例化和初始化对象。它们都需要在类定义中添加

对象初始化器提供了一种简化代码的方式，可以合并对象的实例化和初始化。集合初

13.6.1 对象初始化器

考虑下面的简单类定义：

```
public class Curry
{
    public string MainIngredient { get; set; }
    public string Style { get; set; }
    public int Spiciness { get; set; }
}
```

这个类有3个属性，用第10章介绍的自动属性语法来定义。如果希望实例化和初始化

```
Curry tastyCurry = new Curry();  
tastyCurry.MainIngredient = "panir tikka";  
tastyCurry.Style = "jalfrezi";  
tastyCurry.Spiciness = 8;
```

如果类定义中未包含构造函数，这段代码就使用C#编译器提供的默认无参数构造函数

```
public class Curry  
{  
    public Curry(string mainIngredient, string style,  
  
        int spiciness)  
  
  
    {  
  
  
        MainIngredient = mainIngredient;
```

```
Style = style;
```

```
Spiciness = spiciness;
```

```
}
```

```
...
```

```
}
```

这样就可以编写代码，把实例化和初始化合并起来：

```
Curry tastyCurry = new Curry("panir tikka", "jalfrezi", 8);
```

这段代码工作得很好，但它会强制使用Curry类的代码使用这个构造函数，这将阻止

```

public class Curry
{
    public Curry() {}

    ...
}

```

现在可以用任意方式来实例化和初始化Curry类，但已在最初的类定义中添加几行代

进入对象初始化器（`object initializer`），这是不必在类中添加额外的代码（`initializer`）

```

<ClassName

><variableName

> = new<ClassName

>
{

```

```

        <propertyOrField1

> = <value1

>,
        <propertyOrField2

> = <value2

>,
        ...
        <propertyOrFieldN

> = <valueN

>
};

```

例如，重写前面的代码，实例化和初始化一个Curry类型的对象，如下所示：

```
Curry tastyCurry = new Curry
{
    MainIngredient = "panir tikka",
    Style = "jalfrezi",
    Spiciness = 8
};
```

我们常常可以把这样的代码放在一行上，而不会严重影响可读性。

使用对象初始化器时，不必显式调用类的构造函数。如果像上述代码那样省略构造函数

如果要用对象初始化器进行初始化的属性比本例中使用的简单类型复杂，可以使用嵌

```
Curry tastyCurry = new Curry
{
    MainIngredient = "panir tikka",
    Style = "jalfrezi",
    Spiciness = 8,
    Origin = new Restaurant
    {
        Name = "King's Balti",
```

```
        Location = "York Road",  
        Rating = 5  
    }  
};
```

这里初始化了一个`Restaurant`类型（这里没有列出）的`Origin`属性。代码初始化了

注意，对象初始化器没有替代非默认的构造函数。在实例化对象时，可以使用对象初

另外，在上面的示例中，使用嵌套的对象初始化器和使用构造函数还有一个不太容易

13.6.2 集合初始化器

第5章描述了如何使用如下语法，用值来初始化数组：

```
int[] myIntArray = new int[5] { 5, 9, 10, 2, 99 };
```

这是一种合并实例化和初始化数组的简捷方式。集合初始化器只是把这个语法扩展到


```
List<int> myIntCollection = new List<int> { 5, 9, 10, 2, 99 };
```

通过合并对象和集合初始化器，就可以用简洁的代码来配置集合了。下面的代码：

```
List<Curry> curries = new List<Curry>();  
curries.Add(new Curry("Chicken", "Pathia", 6));  
curries.Add(new Curry("Vegetable", "Korma", 3));  
curries.Add(new Curry("Prawn", "Vindaloo", 9));
```

可以用如下代码替换：

```
List<Curry> moreCurries = new List<Curry>  
{  
    new Curry  
    {  
        MainIngredient = "Chicken",  
        Style = "Pathia",  
        Spiciness = 6  
    },  
    new Curry  
    {
```

```
        MainIngredient = "Vegetable",  
        Style = "Korma",  
        Spiciness = 3  
    },  
    new Curry  
    {  
        MainIngredient = "Prawn",  
        Style = "Vindaloo",  
        Spiciness = 9  
    }  
};
```

这非常适合于主要用于数据表示的类型，因此，集合初始化和本书后面介绍的LINQ

下面的示例说明了如何使用对象和集合初始化器。

试一试：使用初始化器：**Ch13Ex04**

(1) 在C:\BegVCSharp\Chapter13目录中创建一个新的控制台应用程序Ch13Ex

(2) 在Solution Explorer窗口中右击项目名称，选择Add Existing Item选

(3) 在C:\BegVCSharp\Chapter12\Ch12Ex04\Ch12Ex04目录中选择Animal

(4) 修改所添加文件中的名称空间声明，如下所示：

```
namespace Ch13Ex04
```

(5) 删除Cow、Chicken和SuperCow类的构造函数。

(6) 修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)
{
    Farm<Animal> farm = new Farm<Animal>

    {
```

```
new Cow { Name="Lea" },
```

```
new Chicken { Name="Noa" },
```

```
new Chicken(),
```

```
new SuperCow { Name="Andrea" }
```

```
};
```

```
farm.MakeNoises();
```

```
ReadKey();  
  
}
```

(7) 生成应用程序，会得到如图13-10所示的生成错误。因为在Farm类中没有Add

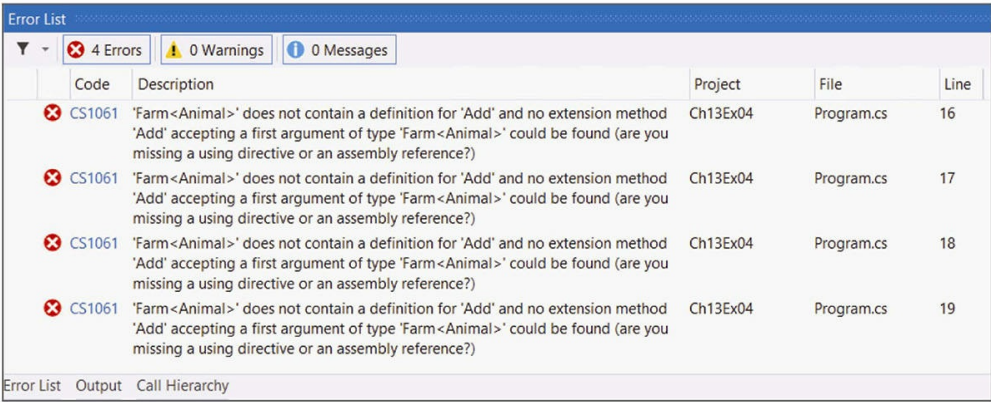


图13-10

(8) 给Farm.cs添加如下代码:

```
public class Farm<T> : IEnumerable<T> where T : Animal
{
    public void Add(T animal) => animals.Add(animal);

    ...
}
```

(9) 运行应用程序，结果如图13-11所示。

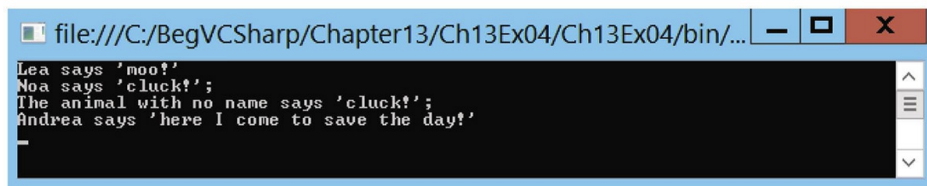


图13-11

示例的说明

这个示例合并了对象和集合初始化器，用一个步骤创建并填充了一个对象集合。它使

首先，给派生于**Animal**基类的类删除构造函数。可以删除这些构造函数，是因为它们

```
public Animal()  
{  
    name = "The animal with no name";  
}
```

但是，对象初始化器与派生于**Animal**类的类一起使用时，初始化器设置的任何属性都

其次，必须给**Farm**类添加**Add()**方法，否则会得到如下形式的一系列编译错误：

```
'Ch13Ex04.Farm<Ch13Ex04.Animal>' does not contain a definition
```

这个错误显示出了集合初始化器的部分底层功能。在后台，编译器为在集合初始化器

还可以修改示例中的代码，为**Animals**属性提供一个嵌套的初始化器，如下所示：

```

static void Main(string[] args)
{
    Farm<Animal> farm = new Farm<Animal>
    {
        Animals =

        {

            new Cow { Name="Lea" },
            new Chicken { Name="Noa" },
            new Chicken(),
            new SuperCow { Name="Andrea" }
        }

    };
    farm.MakeNoises();
    ReadKey();
}

```

有了此代码，就不需要为**Farm**类提供**Add()**方法了。这个技巧适用于包含多个集合的



13.7 类型推理

本书前面介绍过C#是一种强类型化的语言，这表示每个变量都有固定的类型，只能用

```
<type
```

```
><varName
```

```
>;
```

```
<type
```

```
><varName
```

```
> = <value
```

```
>;
```

下面的代码显示了变量<varName>的类型:

```
int myInt = 5;  
WriteLine(myInt);
```

将鼠标指针停放在变量标识符上，IDE就会显示该变量的类型，如图13-12所示。

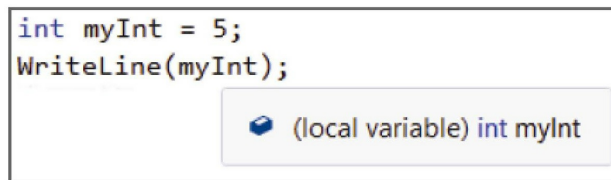


图13-12

C# 3引入了新关键字**var**，它可以替代前面代码中的**type**:

var

`<varName`

`> = <value`

`>;`

在这行代码中，变量`<varName`
>隐式地类型化为`<value`
>的类型。注意，类型的名称并不是`var`。在下面的代码中：

```
var myVar = 5;
```

`myVar`是`int`类型的变量，而不是`var`类型的变量，如图13-13所示，IDE也显示了其

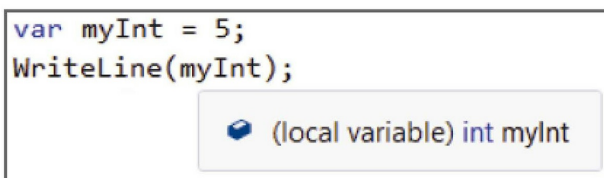


图13-13

这是非常重要的一点。使用**var**时，并不是声明了一个没有类型的变量，也不是声明

注意：

.NET 4引入的动态类型扩展了C#是强类型化语言的定义，参见本章后面的“动态查找”

如果编译器不能确定用**var**声明的变量类型，代码就无法编译。因此，在用**var**声明变

```
var myVar;
```

var关键字还可以通过数组初始化器来推断数组的类型：

```
var myArray = new[] { 4, 5, 2 };
```

在这行代码中，`myArray`类型被隐式地设置为`int[]`。在采用这种方式隐式指定数组

- 相同的类型
- 相同的引用类型或空
- 所有元素的类型都可以隐式地转换为一个类型

如果应用最后一条规则，元素可以转换的类型就称为数组元素的最佳类型。如果这个

```
var myArray = new[] { 4, "not an int", 2 };
```

还要注意数字值从来都不会解释为可空类型，所以下面的代码无法编译：

```
var myArray = new[] { 4, null, 2 };
```

但可以使用标准的数组初始化器，使如下代码编译：

```
var myArray = new int?[] { 4, null, 2 };
```

最后一点要说明的是，标识符**var**并非不能用于类名。这意味着，如果代码在其作用域

类型推理功能本身并不是很有效，因为在本节前面的代码中，它只会使事情更复杂。

13.8 匿名类型

在编写程序一段时间后，会发现我们要花费很多时间为数据表示创建简单、乏味的类

```
public class Curry
{
    public string MainIngredient { get; set; }
    public string Style { get; set; }
    public int Spiciness { get; set; }
}
```

这个类什么也没做，只是存储结构化数据。在数据库或电子表格中，可以把这个类看

这是类完全可以接受的一种用法，但编写这些类的代码比较单调，对底层数据模式的

匿名类型（`anonymous type`）是简化这个编程模型的一种方式。其理念是使用C#的

可按如下方式实例化前面的Curry类型：


```
Curry curry = new Curry
{
    MainIngredient = "Lamb",
    Style = "Dhansak",
    Spiciness = 5
};
```

也可以使用匿名类型，如下所示：

var

```
curry = new
{
    MainIngredient = "Lamb",
    Style = "Dhansak",
    Spiciness = 5
};
```

这里有两个区别。第一，使用了**var**关键字。这是因为匿名类型没有可以使用的标识符。

IDE检测到匿名类型定义后，会相应地更新IntelliSense。通过前面的声明，可以



图13-14

其中，变量curry的类型是'a。显然，不能在代码中使用这个类型——它甚至不是合法

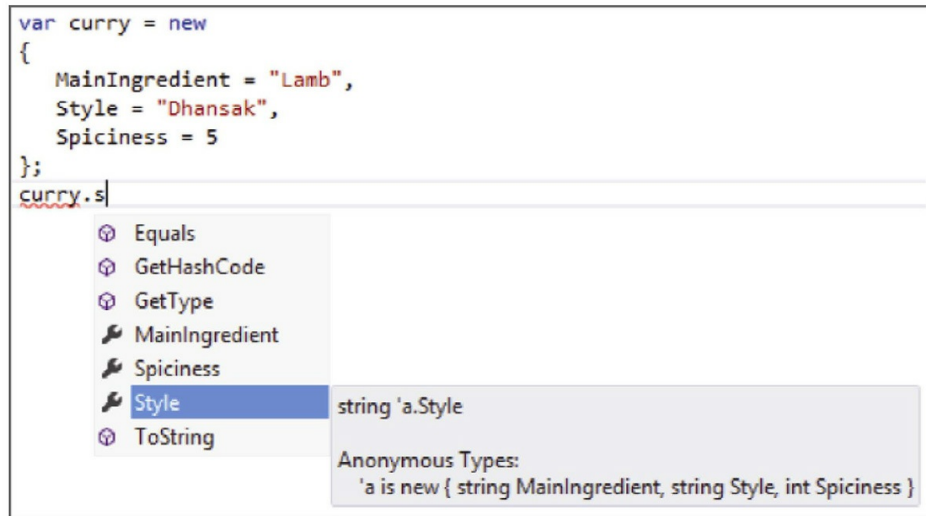


图13-15

注意，这里显示的属性定义为只读属性。这表示，如果要在数据存储对象中修改属性

还实现了匿名类型的其他成员，如下面的示例所示。

试一试：使用匿名类型：**Ch13Ex05\Program.cs**

(1) 在C:\BegVCSharp\Chapter13目录下创建一个新的控制台应用程序Ch13Ex

(2) 修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)
{
    var curries = new[]

{

    new { MainIngredient = "Lamb", Style = "Dhansak", Spicines

    new { MainIngredient = "Lamb", Style = "Dhansak", Spicines

    new { MainIngredient = "Chicken", Style = "Dhansak", Spici
```

```
};
```

```
WriteLine(curries[0].ToString());
```

```
WriteLine(curries[0].GetHashCode());
```

```
WriteLine(curries[1].GetHashCode());
```

```
WriteLine(curries[2].GetHashCode());
```

```
WriteLine(curries[0].Equals(curries[1]));
```

```
WriteLine(curries[0].Equals(curries[2]));
```

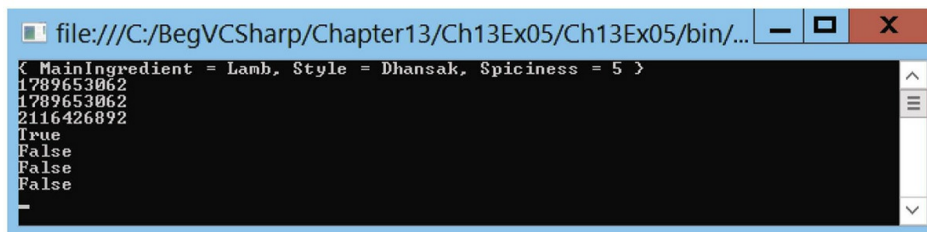
```
WriteLine(curries[0] == curries[1]);
```

```
WriteLine(curries[0] == curries[2]);
```

```
ReadKey();
```

```
}
```

(3) 运行应用程序，结果如图13-16所示。



```
file:///C:/BegVCSharp/Chapter13/Ch13Ex05/Ch13Ex05/bin/...
< MainIngredient = Lamb, Style = Dhansak, Spiciness = 5 >
1789653062
1789653062
2116426892
true
false
false
false
_
```

图13-16

示例的说明

这个示例创建了一个匿名类型对象的数组，然后使用它测试匿名类型提供的成员。创

```
var curries = new[]  
{  
    new { MainIngredient = "Lamb", Style = "Dhansak", Spiciness  
        ...  
};
```

这段代码通过本节和前面“类型推理”一节中介绍的语法，使用了隐式类型化为匿名类

创建这个数组后，代码首先输出在匿名类型上调用ToString()的结果：

```
WriteLine(curries[0].ToString());
```

输出结果如下：

```
{ MainIngredient = Lamb, Style = Dhansak, Spiciness = 5 }
```

匿名类型上的`ToString()`的实现输出了为该类型定义的每个属性的值。

接着，代码在数组的3个对象上分别调用`GetHashCode()`：

```
WriteLine(curries[0].GetHashCode());  
WriteLine(curries[1].GetHashCode());  
WriteLine(curries[2].GetHashCode());
```

`GetHashCode()`执行时，应根据对象的状态为对象返回一个唯一的整数。数组中的i

```
1789653062
```

```
1789653062
```

```
2116426892
```


接着调用`Equals()`方法比较第一个对象和第二个对象，再比较第一个对象和第三个对象。

```
WriteLine(curries[0].Equals(curries[1]));  
WriteLine(curries[0].Equals(curries[2]));
```

结果如下：

True

False

匿名类型上的`Equals()`的实现比较对象的状态，如果一个对象的每个属性值都与另一个对象的每个属性值相等，则返回`true`，否则返回`false`。

但使用`==`运算符不会得到这样的结果。如前几章所述，`==`运算符比较对象引用。最后，

```
WriteLine(curries[0] == curries[1]);  
WriteLine(curries[0] == curries[2]);
```

`curries`数组中的每一项都引用匿名类型的不同实例，所以在两种情况下结果都是`false`。

False

False

有趣的是，在创建匿名类型的实例时，编译器会注意到，参数是相同的，所以创建同



13.9 动态查找

如前所述，`var`关键字本身并不是一个类型，所以并没有违反C#的“强类型化”方法论

引入动态变量的主要目的是在许多情况下，希望使用C#处理另一种语言创建的对象。

```
ScriptObject jsObj = SomeMethodThatGetsTheObject();  
int sum = Convert.ToInt32(jsObj.Invoke("Add", 2, 3));
```

`ScriptObject`类型（这里不深入探讨）提供了一种访问JavaScript对象的方式，

```
int sum = jsObj.Add(2, 3);
```

动态查找功能改变了这一切，它允许编写上述代码，但如下面几节所述，这个功能是

另一个可使用动态查找功能的情形是处理未知类型的C#对象。这听起来似乎很古怪，

在后台，动态查找功能由Dynamic Language Runtime（动态语言运行库，DLR）

动态类型

C# 4引入了dynamic关键字，以用于定义变量。例如：

```
dynamic myDynamicVar;
```

与前面介绍的var关键字不同，的确存在动态类型，所以在声明myDynamicVar时，

注意：

动态类型不同寻常之处在于，它仅在编译期间存在，在运行期间它会被System. Obj

一旦有了动态变量，就可以继续访问其成员（这里没有列出实际获取变量值的代码）

```
myDynamicVar.DoSomething("With this!");
```

无论myDynamicVar实际包含什么值，这行代码都会编译。但是，如果所请求的成员


实际上，像这样的代码提供了一个应在运行期间应用的“处方”。检查myDynamicVar

这最好举例说明。

警告：

下面的示例仅用于演示！一般情况下，应仅在动态类型是唯一选项时才使用它们，例如

试一试：使用动态类型：**Ch13Ex06\Program.cs**



(1) 在C:\BegVCSharp\Chapter13目录中创建一个新的控制台应用程序Ch13Ex

(2) 修改Program.cs中的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.CSharp.RuntimeBinder;
```

```
namespace Ch13Ex06
```

```
{
```

```
    class MyClass1
```

```
    {
```

```
public int Add(int var1, int var2) => var1 + var2;
```

```
}
```

```
class MyClass2 {}
```

```
class Program
```

```
{
```

```
    static int callCount = 0;
```

```
    static dynamic GetValue()
```

```
{
```

```
if (callCount++ == 0)
```

```
{
```

```
    return new MyClass1();
```

```
}
```

```
    return new MyClass2();
```

```
}
```

```
static void Main(string[] args)
```

```
{
```



```
try
```

```
{
```

```
    dynamic firstResult = GetValue();
```

```
    dynamic secondResult = GetValue();
```

```
    WriteLine($"firstResult is: {firstResult.ToString()}")
```

```
    WriteLine($"secondResult is: {secondResult.ToString()})
```

```
    WriteLine($"firstResult call: {firstResult.Add(2, 3)}")
```

```
        WriteLine($"secondResult call: {secondResult.Add(2, 3)}");  
    }  
}
```

```
catch (RuntimeBinderException ex)
```

```
{
```

```
    WriteLine(ex.Message);
```

```
}
```

```
ReadKey();
```

```
}  
}  
}
```

(3) 运行应用程序，结果如图13-17所示。

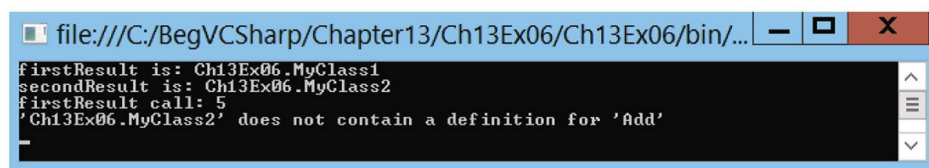


图13-17

示例的说明

这个示例使用一个方法返回两个类型的对象中的一个，以获取动态值，再尝试使用所

首先，为包含`RuntimeBindingException`异常的名称空间添加一条`using`语句：

```
using Microsoft.CSharp.RuntimeBinder;
```

接着定义两个类`MyClass1`和`MyClass2`，其中`MyClass1`包含`Add()`方法，而`MyClass2`包含`GetValue()`方法。

```
class MyClass1
{
    public int Add(int var1, int var2) => var1 + var2;
}
class MyClass2
{
}
```

还要给`Program`类添加一个字段（`callCount`）和一个方法（`GetValue()`），以获取`MyClass1`的实例。

```
static int callCount = 0;
static dynamic GetValue()
{
    if (callCount++ == 0)
    {
        return new MyClass1();
    }
}
```

```

    }
    return new MyClass2();
}

```

使用一个简单的调用计数器，这样，第一次调用这个方法时，返回MyClass1的一个实例；

接着，Main()中的代码调用GetValue()方法两次，再尝试在返回的两个值上依次调

```

static void Main(string[] args)
{
    try
    {
        dynamic firstResult = GetValue();
        dynamic secondResult = GetValue();
        WriteLine($"firstResult is: {firstResult.ToString()}")
        WriteLine($"secondResult is: {secondResult.ToString()}")
        WriteLine($"firstResult call: {firstResult.Add(2, 3)}")
        WriteLine($"secondResult call: {secondResult.Add(2, 3)}")
    }
    catch (RuntimeBinderException ex)
    {
        WriteLine(ex.Message);
    }
    ReadKey();
}

```

```
}
```

可以肯定，调用`secondResult.Add()`时会抛出一个异常，因为在`MyClass2`上不存

`dynamic`关键字也可用于其他需要类型名的地方，例如方法参数。`Add()`方法可以重

```
public int Add(dynamic
```

```
var1, dynamic
```

```
var2) => var1 + var2;
```

这对结果没有任何影响。在这个例子中，传送给`var1`和`var2`的值在运行期间检查，

```
WriteLine("firstResult call: {0}", firstResult.Add("2"
```

```
, 3));
```

异常消息就如下所示：

```
Cannot implicitly convert type 'string' to 'int'
```

从这里获得的教训是动态类型是非常强大的，但有一个警告。如果用强类型代替动态

13.10 高级方法参数

C# 4扩展了定义和使用方法参数的方式。这主要是为了响应使用外部定义的接口时出

```
RemoteCall(var1, var2, null, null, null, null, null);
```

在这行代码中，`null`值表示什么并不明显，或者它们为什么省略并不清楚。

也许，理想情况下，这个`RemoteCall()`方法有多个重载版本，其中一个重载版本仅

```
RemoteCall(var1, var2);
```

但是，这需要更多带其他参数组合的方法，这本身就会带来更多问题（要维护更多的

`Visual Basic`等语言以另一种方式处理这种情况，即允许使用命名参数和可选参数

下面几节介绍如何使用这些新的参数类型。

13.10.1 可选参数

调用方法时，常常给某个参数传送相同的值。例如，这可能是一个布尔值，以控制方

```
public List<string> GetWords(string sentence, bool capitalizeWords)
{
    ...
}
```

无论给`capitalizeWords`参数传送什么值，这个方法都会返回一系列`string`值，在

```
List<string> words = GetWords(sentence, false);
```

为将这种方式变成“默认”方式，可以声明第二个方法，如下所示：

```
public List<string> GetWords(string sentence) => GetWords(sentence, false);
```

这个方法调用第二个方法，并给`capitalizeWords`传送值`false`。

这么做没有任何错误，但可以想象在使用更多的参数时，这种方式会非常复杂。

另一种方式是吧`capitalizeWords`参数变成可选参数。这需要在方法定义中为参数

```
public List<string> GetWords(string sentence, bool capitalizeWords)
{
    ...
}
```

如果以这种方式定义方法，就可以提供一个或两个参数，只有希望`capitalizeWords`

1. 可选参数的值

如上一节所述，为方法定义可选参数的语法如下所示：

<parameterType>

><parameterName

> = <defaultValue

>

对于<defaultValue>的值，存在一些限制。默认值必须是字面值、常量值或者默认

```
public bool CapitalizationDefault;
```

```
public List<string> GetWords(string sentence,  
    bool capitalizeWords = CapitalizationDefault  
  
)  
{  
    ...  
}
```

为使上述代码可以工作，`CapitalizationDefault`值必须定义为常量：

```
public const bool CapitalizationDefault = false  
  
;
```

这是否有意义取决于具体情形，大多数情况下，最好提供一个字面值，就像上一节那

2. `Optional`特性

除了前面小节中描述的语法，还可以使用`Optional`特性定义可选参数，如下所示：

```
[Optional]<parameterType  
  
><parameterName
```

>

此特性包含在`System.Runtime.InteropServices`名称空间中。注意，如果使用

3. 可选参数的顺序

使用可选值时，它们必须位于方法的参数列表末尾。没有默认值的参数不能放在有默

因此下面的代码是非法的：

```
public List<string> GetWords(bool capitalizeWords = false, str  
  
{  
    ...  
}
```

其中，`sentence`是必选参数，因此必须放在可选参数`capitalizedWords`的前面。

13.10.2 命名参数

使用可选参数时，可能发现某个方法有几个可选参数，但可能只想给第三个可选参数

C# 4引入了命名参数（`named parameters`），它允许指定要使用哪个参数。这不

```
MyMethod(  
    <param1Name  
  
>:<param1Value  
  
>,  
    ...  
    <paramNName  
  
>:<paramNValue
```

```
>);
```

参数的名称是在方法定义中使用的变量名。

只要命名参数存在，就可以采用这种方式指定需要的任意多个参数，而且参数的顺序

可以仅给方法调用中的某些参数使用命名参数。当方法签名中有多个可选参数和一些

```
MyMethod(  
    requiredParameter1Value,  
    optionalParameter5: optionalParameter5Value);
```

但注意，如果混合使用命名参数和位置参数，就必须先包含所有的位置参数，其后是

```
MyMethod(  
    optionalParameter5: optionalParameter5Value,  
    requiredParameter1: requiredParameter1Value);
```

此时，必须包含所有必选参数的值。

下面的示例介绍了如何使用命名参数和可选参数。

试一试：使用命名参数和可选参数：**Ch13Ex07**

(1) 在目录C:\BegVCSharp\Chapter13中创建一个新的控制台应用程序Ch13Ex

(2) 在项目中添加一个类WordProcessor，修改其代码，如下所示：

```
public static
```

```
class WordProcessor
```

```
{
```

```
    public static List<string> GetWords(
```



```
string sentence,
```

```
bool capitalizeWords = false,
```

```
bool reverseOrder = false,
```

```
bool reverseWords = false)
```

```
{
```

```
List<string> words = new List<string>(sentence.Split(' '))
```

```
if (capitalizeWords)
```

```
words = CapitalizeWords(words);
```

```
if (reverseOrder)
```

```
words = ReverseOrder(words);
```

```
if (reverseWords)
```

```
words = ReverseWords(words);
```

```
return words;
```

```
}
```

```
private static List<string> CapitalizeWords(List<string> wor
```

```
{
```

```
    List<string> capitalizedWords = new List<string>();
```

```
    foreach (string word in words)
```

```
    {
```

```
        if (word.Length == 0)
```

```
            continue;
```

```
if (word.Length == 1)
```

```
capitalizedWords.Add(
```

```
word[0].ToString().ToUpper());
```

```
else
```

```
capitalizedWords.Add(
```

```
word[0].ToString().ToUpper()
```

```
    + word.Substring(1));
```

```
}
```

```
return capitalizedWords;
```

```
}
```

```
private static List<string> ReverseOrder(List<string> word
```

```
{
```

```
    List<string> reversedWords = new List<string>();
```

```
for (int wordIndex = words.Count - 1;
```

```
wordIndex >= 0; wordIndex--)
```

```
reversedWords.Add(words[wordIndex]);
```

```
return reversedWords;
```

```
}
```

```
private static List<string> ReverseWords(List<string> words)
```

```
{
```

```
List<string> reversedWords = new List<string>();
```

```
foreach (string word in words)
```

```
    reversedWords.Add(Reword(word));
```

```
return reversedWords;
```

```
}
```

```
private static string Reword(string word)
```

```
{
```

```
StringBuilder sb = new StringBuilder();
```

```
for (int characterIndex = word.Length - 1;
```

```
characterIndex >= 0; characterIndex--)
```

```
sb.Append(word[characterIndex]);
```

```
return sb.ToString();
```

```
}
```



```
}
```

(3) 修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)
{
    string sentence = "his gaze against the sweeping bars has "

    + "grown so weary";

    List<string> words;

    words = WordProcessor.GetWords(sentence);

    WriteLine("Original sentence:");
```

```
foreach (string word in words)
```

```
{
```

```
    Write(word);
```

```
    Write(' ');
```

```
}
```

```
WriteLine('\n');
```

```
words = WordProcessor.GetWords(
```

`sentence,`

`reverseWords: true,`

`capitalizeWords: true);`

`WriteLine("Capitalized sentence with reversed words:");`

`foreach (string word in words)`

`{`

`Write(word);`

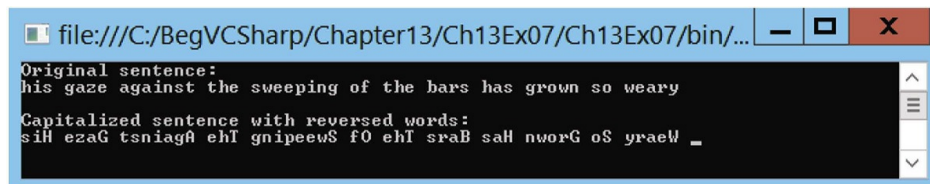
```
Write(' ');
```

```
}
```

```
ReadKey();
```

```
}
```

(4) 运行应用程序，结果如图13-18所示。



```
file:///C:/BegVCSharp/Chapter13/Ch13Ex07/Ch13Ex07/bin/...
Original sentence:
his gaze against the sweeping of the bars has grown so weary
Capitalized sentence with reversed words:
siH ezaG tsniagA ehl gnipeewS fO ehl sraB saH nworG oS yraeW _
```

图13-18

示例的说明

这个示例创建了一个执行一些简单的字符串处理的实用类，再使用这个类修改一个字

```
public static List<string> GetWords(  
    string sentence,  
    bool capitalizeWords = false,  
    bool reverseOrder = false,  
    bool reverseWords = false)  
{  
    ...  
}
```

这个方法返回`string`值的一个集合，每个`string`值都是初始输入的一个单词。根据



注意：

这里并未深入探讨**WordProcessor**类的功能，读者可以自己研究它的代码，考虑一下

调用这个方法时，只使用了两个可选参数，第三个参数（**reverseOrder**）使用其默

```
words = WordProcessor.GetWords(  
    sentence,  
    reverseWords: true,  
    capitalizeWords: true);
```

还要注意，所指定的两个参数的顺序与定义它们的顺序不同。

最后要注意的是，处理带有可选参数的方法时，使用**IntelliSense**会非常方便。输

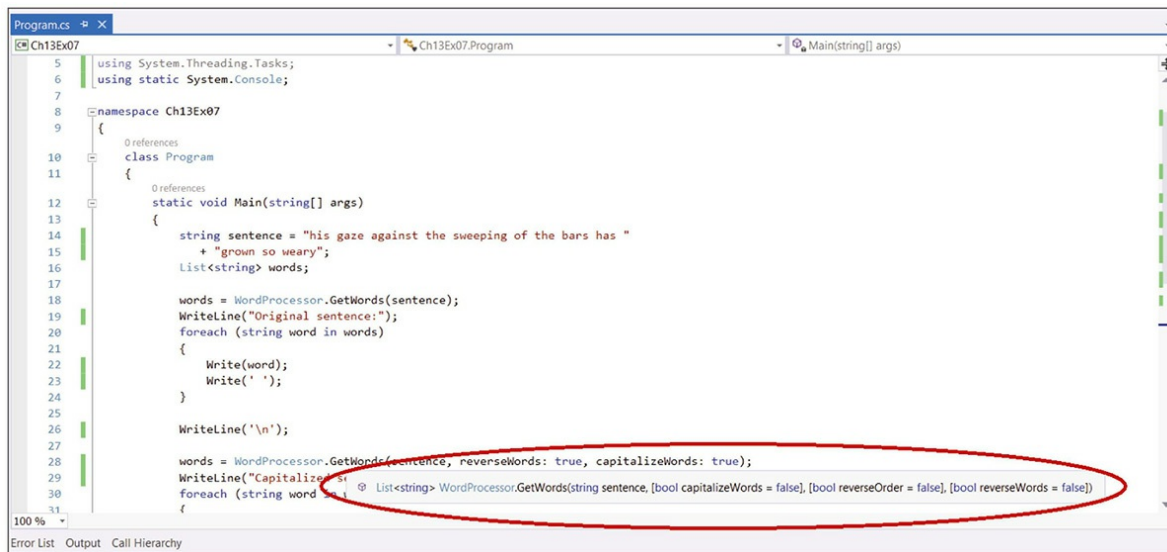


图13-19

这是一个非常有用的工具提示，它不仅显示了可用参数的名称，还显示了可选参数的

13.11 Lambda表达式

Lambda表达式是C# 3.0引入的一个结构，可用于简化C#编程的某些方面，在与LINQ

13.11.1 复习匿名方法

本章前面学习了匿名方法，这是提供的内联（`inline`）方法，否则就需要使用委托来

（1）定义一个事件处理方法，其返回类型和参数匹配要订阅的事件需要的委托的返回

（2）声明一个委托类型的变量，用于事件。

（3）把委托变量初始化为委托类型的实例，该实例指向事件处理方法。

（4）把委托变量添加到事件的订阅者列表中。

实际上，这个过程会比上述简单一些，因为一般不使用变量来存储委托，只在订阅事

前面使用的代码就属于这种情况，如下所示：

```
Timer myTimer = new Timer(100);  
myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

这段代码订阅了Timer对象的Elapsed事件。这个事件使用委托类型ElapsedEvent

实际上， C#编译器可以通过方法组语法，用更少的代码获得相同的结果：

```
myTimer.Elapsed += WriteChar;
```

C#编译器知道Elapsed事件需要的委托类型，所以可以填充该类型。但大多数情况下

(1) 使用内联的匿名方法，该匿名方法的返回类型和参数匹配所订阅事件需要的委托

用delegate关键字定义内联的匿名方法：

```
myTimer.Elapsed +=  
    delegate(object source, ElapsedEventArgs e)  
    {  
        WriteLine("Event handler called after {0} milliseconds.",  
            (source as Timer).Interval);  
    };
```

这段代码像单独使用事件处理程序一样正常工作。主要区别是这里使用的匿名方法对

13.11.2 把Lambda表达式用于匿名方法

下面看一下Lambda表达式。Lambda表达式是简化匿名方法的语法的一种方式。实际

```
myTimer.Elapsed += (source, e) => WriteLine("Event handler cal  
    $"{{(source as Timer).Interval}} milliseconds.");
```

这段代码初看上去有点让人摸不着头脑（除非很熟悉所谓的函数化编程语言，如Lisp）

- 放在括号中的参数列表（未类型化）
- `=>`运算符
- C#语句

使用本章前面“匿名类型”一节中介绍的逻辑，从上下文中推断出参数的类型。`=>`运算符

编译器会提取这个Lambda表达式，创建一个匿名方法，其工作方式与上一节中的匿名方法

为说明Lambda表达式中的内容，下面列举一个例子。

试一试：使用简单的Lambda表达式：**Ch13Ex08\Program.cs**

(1) 在C:\BegVCSharp\Chapter13目录中创建一个新的控制台应用程序Ch13Ex

(2) 修改Program.cs中的代码，如下所示：

```
namespace Ch13Ex08
{
    delegate int TwoIntegerOperationDelegate(int paramA, int paramB);

    class Program
    {
        static void PerformOperations(TwoIntegerOperationDelegate delegate)
        {
            for (int paramAVal = 1; paramAVal <= 5; paramAVal++)
            {
                for (int paramBVal = 1; paramBVal <= 5; paramBVal++)
                {
                    int result = delegate(paramAVal, paramBVal);
                    Console.WriteLine($"paramAVal: {paramAVal}, paramBVal: {paramBVal}, result: {result}");
                }
            }
        }
    }
}
```

```
for (int paramBVal = 1; paramBVal<= 5; paramBVal++)
```

```
{
```

```
    int delegateCallResult = del(paramAVal, paramBVal);
```

```
    write($"{paramAVal}, " +
```

```
        $"{paramBVal})={delegateCallResult}");
```

```
    if (paramBVal != 5)
```

```
{
```

```
Write(", ");
```

```
}
```

```
}
```

```
WriteLine();
```

```
}
```

```
}
```

```
static void Main(string[] args)
{
    WriteLine("f(a, b) = a + b:");

    PerformOperations((paramA, paramB) => paramA + paramB);

    WriteLine();

    WriteLine("f(a, b) = a * b:");

    PerformOperations((paramA, paramB) => paramA * paramB);

    WriteLine();

    WriteLine("f(a, b) = (a - b) % b:");
```

```
PerformOperations((paramA, paramB) => (paramA - paramB)
```

```
% paramB);
```

```
ReadKey();
```

```
}
```

```
}
```

```
}
```

(3) 运行应用程序，结果如图13-20所示。


```
file:///C:/BegVCSharp/Chapter13/Ch13Ex08/Ch13Ex08/bin/...
f(a, b) = a + b:
f(1,1)=2, f(1,2)=3, f(1,3)=4, f(1,4)=5, f(1,5)=6
f(2,1)=3, f(2,2)=4, f(2,3)=5, f(2,4)=6, f(2,5)=7
f(3,1)=4, f(3,2)=5, f(3,3)=6, f(3,4)=7, f(3,5)=8
f(4,1)=5, f(4,2)=6, f(4,3)=7, f(4,4)=8, f(4,5)=9
f(5,1)=6, f(5,2)=7, f(5,3)=8, f(5,4)=9, f(5,5)=10

f(a, b) = a * b:
f(1,1)=1, f(1,2)=2, f(1,3)=3, f(1,4)=4, f(1,5)=5
f(2,1)=2, f(2,2)=4, f(2,3)=6, f(2,4)=8, f(2,5)=10
f(3,1)=3, f(3,2)=6, f(3,3)=9, f(3,4)=12, f(3,5)=15
f(4,1)=4, f(4,2)=8, f(4,3)=12, f(4,4)=16, f(4,5)=20
f(5,1)=5, f(5,2)=10, f(5,3)=15, f(5,4)=20, f(5,5)=25

f(a, b) = (a - b) % b:
f(1,1)=0, f(1,2)=-1, f(1,3)=-2, f(1,4)=-3, f(1,5)=-4
f(2,1)=0, f(2,2)=0, f(2,3)=-1, f(2,4)=-2, f(2,5)=-3
f(3,1)=0, f(3,2)=1, f(3,3)=0, f(3,4)=-1, f(3,5)=-2
f(4,1)=0, f(4,2)=0, f(4,3)=1, f(4,4)=0, f(4,5)=-1
f(5,1)=0, f(5,2)=1, f(5,3)=2, f(5,4)=1, f(5,5)=0
```

图13-20

示例的说明

这个示例使用Lambda表达式生成函数，用来在两个输入参数上执行指定的处理，并返回结果。

首先定义一个委托类型TwoIntegerOperationDelegate，表示一个方法，该方法接受两个整数参数并返回一个整数。

```
delegate int TwoIntegerOperationDelegate(int paramA, int paramB);
```

在以后定义Lambda表达式时使用这个委托类型。这些Lambda表达式编译为方法，其名称为f(a, b)，其中a和b是输入参数。

接着添加方法PerformOperations(), 它带有一个TwoIntegerOperationDele

```
static void PerformOperations(TwoIntegerOperationDelegate  
{
```

这个方法的含义是, 可给它传送一个委托实例 (或者匿名方法, 或者Lambda表达式,

```
for (int paramAVal = 1; paramAVal<= 5; paramAVal++)  
{  
    for (int paramBVal = 1; paramBVal<= 5; paramBVal++)  
    {  
        int delegateCallResult = del(paramAVal, paramBVal);
```

接着把参数和结果输出到控制台上:

```
Write($"f({paramAVal}, " +  
    $"{paramBVal})={delegateCallResult}");  
if (paramBVal != 5)  
{  
    Write(", ");
```

```
        }  
    }  
    WriteLine();  
}  
}
```

在Main()方法中，创建了3个Lambda表达式，使用它们依次调用PerformOperati

```
WriteLine("f(a, b) = a + b:");  
PerformOperations((paramA, paramB) => paramA + paramB);
```

这里使用的Lambda表达式如下：

```
(paramA, paramB) => paramA + paramB
```

这个Lambda表达式分为3部分：

(1) 参数定义部分。这里有两个参数paramA和paramB。这些参数都是未类型化的，

(2) =>运算符。它把Lambda表达式的参数与表达式体分开。

(3) 表达式体。它指定了一个简单操作：把paramA和paramB加起来。注意，不需

接着，就可以把使用这个Lambda表达式的代码扩展到下面使用匿名方法的代码中：

```
WriteLine("f(a, b) = a + b:");
PerformOperations(delegate(int paramA, int paramB)
{

    return paramA + paramB;

}));
```

其余代码以相同方式使用两个不同的Lambda表达式来执行操作：

```
WriteLine();  
WriteLine("f(a, b) = a * b:");  
PerformOperations((paramA, paramB) => paramA * paramB);  
WriteLine();  
WriteLine("f(a, b) = (a - b) % b:");  
PerformOperations((paramA, paramB) => (paramA - paramB)  
    % paramB);  
ReadKey();
```

最后一个Lambda表达式涉及较多计算，但并不比其他Lambda表达式更复杂。Lambc

13.11.3 Lambda表达式的参数

在前面的代码中，Lambda表达式使用类型推理功能来确定所传送的参数类型。实际

```
(int paramA, int paramB) => paramA + paramB
```

其优点是代码更便于理解，缺点是不够简明灵活。在前面委托类型的示例中，可以通

注意，不能在同一个Lambda表达式中同时使用隐式和显式的参数类型。下面的Lamb

```
(int paramA, paramB) => paramA + paramB
```

Lambda表达式的参数列表始终包含一个用逗号分隔的列表，其中的参数要么都是显式

```
param1 => param1 * param1
```

还可以定义没有参数的Lambda表达式，这使用空括号来表示：

```
() => Math.PI
```

当委托不需要参数，但需要返回一个double值时，就可以使用这个Lambda表达式。

13.11.4 Lambda表达式的语句体

在前面的所有代码中，**Lambda**表达式的语句体都只使用了一个表达式。我们还说明

前一个示例说明了对于语句体中使用的代码而言，返回类型为**void**的委托的要求并不

```
myTimer.Elapsed += (source, e) => WriteLine("Event handler called  
    ${source as Timer}.Interval} milliseconds.");
```

上面的语句不返回任何值，所以它只是执行，其返回值不在任何地方使用。

可将**Lambda**表达式看成匿名方法语法的扩展，所以还可以在**Lambda**表达式的语句体

```
(param1, param2) =>  
{  
    // Multiple statements ahoy!  
}
```

如果使用**Lambda**表达式和返回类型不是**void**的委托类型，就必须用**return**关键字让

```
(param1, param2) =>
{
    // Multiple statements ahoy!
    return returnValue;
}
```

例如，可将前面示例中的如下代码：

```
PerformOperations((paramA, paramB) => paramA + paramB);
```

改写为：

```
PerformOperations(delegate(int paramA, int paramB)
{
    return paramA + paramB;
});
```

另外，也可以把代码改写为：


```
PerformOperations((paramA, paramB) =>
```

```
{  
    return paramA + paramB;  
});
```

这更像是原来的代码，因为它保持了paramA和paramB参数的隐式类型化。

大多数情况下，在使用单一表达式时，Lambda表达式最有用，也最简洁。说实话，

13.11.5 Lambda表达式用作委托和表达式树

前面提到了Lambda表达式和匿名方法的一些区别：Lambda表达式比较灵活，例如，

可采用两种方式来解释Lambda表达式。第一，如本章所述，Lambda表达式是一个委

一般可以把拥有至多8个参数的Lambda表达式表示为如下泛型类型，它们都在System

- `Action`，表示的Lambda表达式不带参数，返回类型是`void`
- `Action<>`，表示的Lambda表达式有至多8个参数，返回类型是`void`
- `Func<>`，表示的Lambda表达式有至多8个参数，返回类型不是`void`

`Action<>`最多有8个泛型类型的参数，分别用于Lambda表达式的8个参数，`Func<>`

例如，下面的Lambda表达式：

```
(int paramA, int paramB) => paramA + paramB
```

可以表示为`Func<int, int, int>`类型的委托，因为它有两个`int`参数，返回类型

第二，可以把Lambda表达式解释为表达式树。表达式树是Lambda表达式的抽象表示

显然这是一个复杂主题，但表达式树对本书后面介绍的LINQ功能至关重要。下面列举

目前并不需要了解太多内容，在本书后面遇到这个功能时，能更好地理解其过程，因

13.11.6 Lambda表达式和集合

学习了Func<>泛型委托后，就可以理解System.Linq名称空间为数组类型提供的一

```
public static TSource Aggregate<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, TSource, TSource> func);
public static TAccumulate Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func);
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func,
    Func<TAccumulate, TResult> resultSelector);
```

与前面的扩展方法一样，这段代码初看上去非常深奥，但如果分解它们，就很容易理

`Applies an accumulator function over a sequence.`

这表示要把一个累加器函数（可以采用**Lambda**表达式的形式提供）应用于集合中从开

在3个重载版本中，最简单的版本只有一个泛型类型，这可从实例参数的类型推理出开

```
int[] myIntArray = { 2, 6, 3 };  
int result = myIntArray.Aggregate(...);
```

这等价于：

```
int[] myIntArray = { 2, 6, 3 };  
int result = myIntArray.Aggregate<int>  
  
(...);
```

这里需要的Lambda表达式可以从扩展方法中推断出来。在这段代码中，类型TSource

```
int[] myIntArray = { 2, 6, 3 };  
int result = myIntArray.Aggregate((paramA, paramB) => paramA +
```

这个调用会使Lambda表达式调用两次，一次使用的参数是paramA=2，paramB=6，

扩展方法Aggregate()的其他两个重载版本是类似的，但可以执行略微复杂的计算，

试一试：使用Lambda表达式和集合：Ch13Ex09\Program.cs

(1) 在C:\BegVCSharp\Chapter13目录中创建一个新的控制台应用程序Ch13Ex

(2) 修改Program.cs中的代码，如下所示：

```
static void Main(string[] args)
{
    string[] curries = { "pathia", "jalfrezi", "korma" };

    WriteLine(curries.Aggregate(

        (a, b) => a + " " + b));

    WriteLine(curries.Aggregate<string, int>(

        0,

        (a, b) => a + b.Length));
```

```
WriteLine(curries.Aggregate<string, string, string>(
```

```
"Some curries:",
```

```
(a, b) => a + " " + b,
```

```
a => a));
```

```
WriteLine(curries.Aggregate<string, string, int>(
```

```
"Some curries:",
```

```
(a, b) => a + " " + b,
```

```
a => a.Length));
```

```
ReadKey();
```

```
}
```

(3) 运行应用程序，结果如图13-21所示。

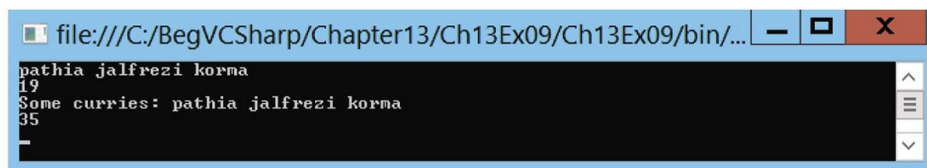


图13-21

示例的说明

这个示例把包含3个元素的字符串数组作为源数据，试验了扩展方法Aggregate()的

首先执行一个简单的串联操作：

```
WriteLine(curries.Aggregate((a, b) => a + " " + b));
```

第一对元素用简单的语法串联成一个字符串。这远非连接字符串的最佳方式，理想情

接着使用Aggregate()函数的第二个重载版本，它有两个泛型类型的参数TSource和TAccumulate。

```
WriteLine(curries.Aggregate<string, int>(
    0,
    (a, b) => a + b.Length));
```

累加器（以及返回值）的类型是int。累加器的值最初设置为种子值0，在对Lambda

之后使用Aggregate()函数的最后一个重载版本，它带有3个泛型类型的参数，与前

```
WriteLine(curries.Aggregate<string, string, string>(
    "Some curries:",
    (a, b) => a + " " + b,
    a => a));
```

即使累加值只是复制到结果中（如本例所示），也必须指定这个方法的最后一个参数

在最后一段代码中，再次使用了Aggregate()的这个版本，但这次使用int类型的返

```
WriteLine(curries.Aggregate<string, string, int>(
    "Some curries:",
    (a, b) => a + " " + b,
    a => a.Length));
```

这个示例没有什么花哨的地方，但演示了如何使用更复杂的扩展方法，其中涉及泛型

13.12 练习

- (1) 编写事件处理程序的代码，这些代码使用了通用语法 (`object sender, EventArgs e`)。
- (2) 修改扑克牌游戏示例，设置流行拉米扑克牌的更有趣的取胜条件。即一个玩家要取得 10 张牌，且其中 5 张牌是同一花色，且 5 张牌是同一数字。
- (3) 为什么不能把对象初始化器用于下面的类？修改这个类，使其能使用对象初始化器。

```
public class Giraffe
{
    public Giraffe(double neckLength, string name)
    {
        NeckLength = neckLength;
    }
}
```

```
Name = name;
```

```
}
```

```
public double NeckLength {get; set;}
```

```
public string Name {get; set;}
```

```
}
```

(4) 判断正误：如果声明一个**var**类型的变量，就可以使用它存储任意对象类型。

(5) 使用匿名类型时，如何比较两个实例，确定它们是否包含相同的数据？

(6) 更正下面扩展方法的代码，其中包含一个错误：

```

public string ToAcronym(this string inputString)
{
    inputString = inputString.Trim();
    if (inputString == "")
    {
        return "";
    }
    string[] inputStringAsArray = inputString.Split(' ');
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i<inputStringAsArray.Length; i++)
    {
        if (inputStringAsArray[i].Length > 0)
        {
            sb.AppendFormat("{0}",
                inputStringAsArray[i].Substring(
                    0, 1).ToUpper());
        }
    }
    return sb.ToString();
}

```

(7) 如何确保练习题 (6) 中的扩展方法可用于客户代码?

(8) 把ToAcronym方法改为一条语句。该代码应确保单词之间包含多个空格的字符

附录A给出了练习答案。

13.13 本章要点

主题	要点
名称空间限定符	为了避免名称空间限定的模糊性，可以使用 <code>::</code> 运算符强制编译器使用已创建好的别名。还可以使用 <code>global</code> 名称空间作为顶级名称空间的别名
定制异常	从根类 <code>Exception</code> 中派生，就可以创建自己的异常类。这是有益的，因为可以更多地控制特定异常的捕获，并允许定制包含在异常中的数据，以高效地处理它
事件处理	许多类都提供了事件，在代码中发生某个触发器时，就会引发这些事件。可为这些事件编写处理程序，在引发事件时执行代码。这种双向通信方式是响应代码的一种良好机制，不必编写可能要轮询对象以获知变化的复杂、令人感到费解的代码
事件定义	可以定义自己的事件类型，这涉及给事件的处理程序创建指定的事件和委托类型。可以使用标准的、无返回类型的委托类型和派生于 <code>System.EventArgs</code> 的定制事件参数，使事件处理程序有多种用途。还可以使用 <code>EventHandler</code> 和 <code>EventHandler<T></code> 委托类型，以便通过更简单的代码来定义事件
匿名方法	为使代码更便于阅读，常可以使用匿名方法来替代完整的事件处理方法。这表示，在添加事件处理程序的地方直接定义要在引发事件时执行的代码，为此需要使用 <code>delegate</code> 关键字
特性	有时，或者是由于所用框架的要求，或者是由于自己的需要，在代码中会用到特性。通过使用 <code>[AttributeName]</code> 语法，可以向类、方法和其他成员添加特性；通过从 <code>System.Attribute</code> 派生，可以创建自己的特性。通过反射可读取特性值
初始化器	可使用初始化器在创建对象或集合的同时初始化它们。这两种初始化器都包括一个放在花括号中的代码块。对象初始化器可以提供一个逗号分隔的属性名/值对列表，来设置属性值。集合初始化器仅需要逗号分隔的值列表。使用对象初始化器时，还可以使用非默认的构造函数
类型推理	声明变量时，使用 <code>var</code> 关键字允许忽略变量的类型。但只有类型可以在编译期间确定时才可以这么做。使用 <code>var</code> 没有违反C#的强类型化规则，

	因为用 var 声明的变量只能有一种类型
匿名类型	对于用于数据存储的许多简单类型，并非必须定义类型。相反，可以使用匿名类型，其成员根据用途来推断。使用对象初始化器语法来定义匿名类型，每个设置的属性都定义为只读属性
动态查找	使用 dynamic 关键字定义 dynamic 类型的变量，可以存储任意值。接着就可以使用一般的属性或方法语法来访问该变量中包含的值的成员，这些成员仅在运行期间检查。如果在运行期间，尝试访问一个不存在的成员，就会抛出一个异常。这种动态的类型化显著简化了访问非 .NET 类型或类型信息不能在编译期间获得的 .NET 类型的语法。但是，在使用动态类型时要谨慎，因为无法在编译期间检查代码。实现 IDynamicMetaObjectProvider 接口，可以控制动态查找的行为
可选的方法参数	我们常常可以定义带许多参数的方法，但其中的许多参数都很少使用。可以提供多个方法重载，而不是强制客户代码为很少使用的参数提供值。另外，也可以把这些参数定义为可选参数（并为未指定值的参数提供默认值）。调用方法的客户代码就可以仅指定需要的参数
命名的方法参数	客户代码可以根据位置或名称（或者根据位置和名称，其中位置参数放在前面）来指定方法的参数。命名的参数可按任意顺序指定。这尤其适用于和可选参数一起使用的场合
Lambda表达式	Lambda 表达式实际上是定义匿名方法的一种快捷方式，而且具有额外的功能，例如隐式的类型化。定义 Lambda 表达式时，需要使用逗号分隔的参数列表（如果没有参数，就使用空括号）、 => 运算符和一个表达式。该表达式可以是放在花括号中的代码块。 Lambda 表达式至多可以有8个参数和一个可选的返回类型， Lambda 表达式可以用 Action 、 Action<> 和 Func<> 委托类型来表示。许多可用于集合的 LINQ 扩展方法都使用 Lambda 表达式参数

第 II 部分 **Windows**编程

➤ 第14章 基本桌面编程

➤ 第15章 高级桌面编程

第14章 基本桌面编程

本章内容：

- 如何使用WPF设计器
- 如何使用Label和TextBlock等控件向用户呈现信息
- 如何使用Button等控件触发事件
- 如何使用TextBox等控件让应用程序的用户输入文本
- 如何使用RadioButton和CheckBox等控件将应用程序的当前状态告知用户，并允许用户修改状态
- 如何使用ListBox和ComboBox等控件显示信息列表
- 如何使用窗格对用户界面进行布局

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2014programming。从该网页的Download Code选项卡中下载Chapter 14 Code后，可以找到与本章示例对应的单独文件。

本书第 I 部分由内而外详细介绍了C#语言的一些知识，但从本章开始，不再介绍编程语言的细节，而将进入图形用户界面（Graphical User Interface，GUI）的世界。

过去10年中，Visual Studio为Windows开发人员提供了多种创建用户界面的方法：Windows Forms是用于创建传统Windows应用程序的基本工具，Windows Presentation Foundation（WPF）则提供更广泛的应用程序类型，并尝试解决Windows Forms中存在的很多问题。从技术角度看，WPF是独立于平台的，例如，WPF的子集Silverlight用于创建交互式Web应用程序，WPF的灵活性由此可见一斑。本章和下一章将讨论如何用WPF创建Windows应用程序，第23章将在此基础上创建通用Windows应用程序。

对于大多数图形Windows应用程序而言，开发核心是窗口设计器。为创建用户界面，将控件从工具箱拖放到窗口中，放在应用程序运行时希望其出现的位置上。而在WPF中，则不完全是这样，原因是用户界面实际上完全由另一种称为“可扩展应用程序标记语言（Extensible Application Markup Language，简称XAML，读作zammel）的语言来编写。在Visual Studio中，两种方式都可行，随着自己更加熟悉WPF，既可以拖曳控件，也可以直接编写XAML代码。

本章将使用Visual Studio WPF设计器为前面章节中编写的纸牌游戏创建很多窗口。Visual Studio中自带了许多拥有广泛功能的控件，本章将用到其中的一部分。利用Visual Studio的设计功能，开发用户界面和处理用户交互变得十分直观，而且充满趣味！由于篇幅所限，本书无法涵盖所有Visual Studio控件。本章要介绍一些最常用的控件，包括标签、文本框、菜单栏和布局面板等控件。

14.1 XAML

XAML是一门使用XML语法的语言，允许以层次化的声明方式将控件添加到用户界面中。也就是说，可以采用XML元素的形式添加控件，并使用XML特性来指定控件属性。也可以使用包含其他控件的控件，这在布局和功能上都是必需的。

注意： 第19章将详细介绍XML。如果此时希望快速了解XML的基础知识，可直接跳到该章，阅读其前几页的内容。

XAML在设计时就考虑到利用当今功能强大的显卡，允许通过DirectX来使用这些显卡提供的所有高级功能。下面列出其中一些功能：

- 浮点坐标和矢量图形，允许在不损失质量的情况下缩放、旋转和转换布局
- 高级2D和3D渲染功能
- 高级字体处理和渲染
- UI对象支持纯色、渐变和纹理填充，并可选择透明度
- 可在任何情形中使用的动画分镜头设计，包括鼠标单击按钮等用户触发的事件
- 可使用可重用的资源来动态设置控件的样式

14.1.1 关注点分离

在过去几年中，维护Windows应用程序的一个问题在于，生成用户界面的代码和基于用户操作执行的代码经常混合在一起。这导致多个开发人员和设计人员难以处理同一个项目。WPF通过两种途径解决这个问题。首先，使用XAML（而不是C#）来描述GUI，GUI因此变得独立于平台，实际上，可在不使用任何代码的情况下渲染XAML。其次，很自然会将C#代码与GUI代码放在不同文件中。Visual Studio使用了“代码隐藏文件”，即能动态链接到XAML文件的C#文件。

由于GUI与代码分离开来，可以创建定制的应用程序来设计GUI，Microsoft已经做到了这一点。Blend for Visual Studio是设计师们为WPF制作GUI时的首选工具。该工具可与Visual Studio加载相同的项目，但Visual Studio主要面向开发人员，而不是设计人员；Expression Blend恰好相反。也就是说，如果有许多设计人员和开发人员参与到大型项目中，他们可以使用各自喜欢的工具共同处理同一个项目，而不必担心无意间影响他人的工作。

14.1.2 XAML基础知识

正如前面介绍的那样，XAML是XML语言，这意味着在XAML较小时，我们可以直接看清代码所要表达的含义。请分析下面这段代码，看你能否理解它所表达的含义：

```
<Window x:Class="Ch14Ex01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2
xmlns:mc="http://schemas.openxmlformats.org/markup-compa
xmlns:local="clr-namespace:WpfApplication1"
mc:Ignorable="d"
Title="Hello World" Height="350" Width="525">
<Grid>
<Button Content="Hello World"
    HorizontalAlignment="Left"
    Margin="220,151,0,0"
    VerticalAlignment="Top"
    Width="75"/>
</Grid>
</Window>

```

上述XAML示例的作用是创建带有一个按钮的窗口。窗口和按钮中会显示Hello World文字。XML允许在一个标签中放置另一个标签，只需要正确地闭合各个标签即可。在XAML中，如果将一个元素放在另一个元素中，前者将成为后者内容的一部分，也就是说Button部分的代码也可以编写为：

```

<Button HorizontalAlignment="Left"
    Margin="220,151,0,0"
    VerticalAlignment="Top"
    Width="75">
    Hello World
</Button>

```

上述代码中，`Button`的`Content`属性被删除掉了，这样，文本就成为`Button`控件的子节点。在XAML中，`Content`可以是任意内容，正如在上述例子中演示的那样：`Button`元素是`Grid`元素的内容，而这个`Grid`元素又是`Window`元素的内容。

绝大多数控件（但不是全部控件）都可以包含内容，并且对内置控件外观的修改只有很少的限制。第15章将详细介绍这部分内容。

1. 名称空间

在上个例子中，`Window`元素是XAML文件的根元素。该元素通常包含一系列名称空间声明。默认情况下，`Visual Studio`设计器中包含两个值得注意的名称空间：

<http://schemas.microsoft.com/winfx/2006/xaml/presentation>和

<http://schemas.microsoft.com/winfx/2006/xaml>。前者是WPF的默认名称空间，其中声明了许多在创建用户界面时可能用到的控件。后者则用于声明XAML语言本身。名称空间并非必须在根标签中声明，不过在这里声明可以保证整个XAML文件范围内都可以方便地访问到这个名称空间中的内容，因此通常没必要将这些声明放到其他位置。

注意：名称空间看起来像是URL，但这是有欺骗性的。实际上它们称为Uniform Resource Identifiers（URI）。URI可以是任意字符串，只要它唯一地标识一个资源即可。Microsoft选择通常用于URL的形式指定URI，但在这里，如果把它们输入浏览器，就不会得到什么结果。

在Visual Studio中新建了一个窗口后，总会默认声明一个presentation名称空间，而XAML语言的名称空间则以xmlns:x形式进行声明。正如Window、Button和Grid标签那样，这样声明之后可以不必再为添加到窗口中的控件添加前缀，但我们指定的语言元素必须标明x前缀。

最后一个十分常见的名称空间是系统名称空间：xmlns:sys="clr-namespace:System;assembly=mscorlib"。该名称空间允许在XAML中直接使用.NET Framework内置的类型。这样做之后，在代码中所写的标记可以显式声明要创建的元素类型。例如，可在标记中声明一个数组，并且表明数组中的成员是字符串：

```
<Window.Resources>
  <ResourceDictionary>
    <x:Array Type="sys:String" x:Key="localArray">
      <sys:String>"Benjamin Perkins"</sys:String>
      <sys:String>"Jacob Vibe Hammer"</sys:String>
      <sys:String>"Job D. Reid"</sys:String>
    </x:Array>
  </ResourceDictionary>
</Window.Resources>
```

2. 代码隐藏文件

尽管XAML是一种强大的用户界面声明方式，但它并不是一门编程语言。如果我们想在界面表现的基础上增加一些功能，则需要使用C#代码。虽然可在XAML中直接嵌入C#代码，但任何时候都不建议将代码和

标记混合在一起，因而本书也不会这么做。本书将要大量用到的是“代码隐藏文件”。它们就是普通C#文件，只不过其名称与XAML文件相同，再加上.cs扩展名。尽管也可以将其命名为其他文件名，但最好遵循上述命名约定。为应用程序创建新窗口时，Visual Studio会自动创建代码隐藏文件，因为它知道我们会为该窗口添加代码。同时，Visual Studio也会在XAML文件的Window标签中添加x:Class属性：

```
<Window x:Class="Ch14Ex01.MainWindow"
```

这条语句告诉编译器，该窗口对应的代码不在一个单独文件中，而在Ch14Ex01.MainWindow类中。因为我们只能指定完全限定的类名，不能指定包含该类的程序集，因此不能把代码隐藏文件放在定义该XAML文件的项目之外。Visual Studio自动将代码隐藏文件与XAML文件放在同一个目录中，因此使用Visual Studio时，我们不必担心发生上述情况。

14.2 动手实践

现在，你已经对WPF的结构有了足够的了解，可以开始亲手实践了。我们一起来了解一下编辑器。首先新建一个WPF项目，方法是选择File|New|Project。在New Project对话框中，导航到Visual C#|Windows下的Clasic Desktop节点，选择项目模板WPF Application，为使这个例子可以在下一个例子中重用，把项目命名为Ch14Ex01。

现在，Visual Studio界面中会显示一个空白窗口，四周则是各种不同的面板。屏幕的主要区域分为两部分。上部为设计视图，用于显示当前设计的窗口的所见即所得（WYSIWYG）外观；下部是XAML视图，用于显示同一窗口的代码。

在设计视图的右侧，是前面项目中已经有所接触的Solution Explorer，以及Properties面板，该面板显示了当前在设计视图和XAML视图中所选内容的相关信息。需要注意，属性面板中显示的内容和XAML视图、设计视图中的选择总是同步的；如果在XAML视图中移动光标，其他两个区域中的选择的内容也会随之自动变化。

设计视图左侧有几个收缩起来的面板，其中之一是工具箱。本章将介绍如何使用工具箱中的各种控件为纸牌游戏创建对话框，因此请将其展开，然后单击其右上角的固定按钮将其固定为展开状态。随后在该面板中展开Common WPF Controls节点。本章将主要介绍此处的大部分控件。

14.2.1 WPF控件

所谓控件，是将程序代码和GUI预先打包到一起，可供重复利用，并创建出复杂的应用程序。控件可以定义自身默认的绘制形式及一系列标准行为。Label、Button和TextBox等控件很容易识别，因为它们它们在Windows应用程序中已经被使用了约20年。其他控件，如Canvas和StackPanel，不显示任何内容，只是用来帮助创建GUI。

自带控件的外观看起来与标准Windows应用程序中的控件是一样的，它们可以按照当前的Windows主题设置绘制自身。不过，所有外观元素都可以高度自定义，只需单击几次鼠标，就可以完全改变这些控件的显示方式。这样的自定义是通过设置控件的属性值来实现的。WPF不仅可以使我们之前所了解到的标准属性，还支持一种新的“依赖属性（dependency property）”。第15章将详细介绍这些属性，现在只需知道许多WPF属性并不仅可以获取和设置值；例如，它们能够将自身的更改告知观察者。

除了可以定义其在屏幕上的外观外，控件中也定义了一些标准行为，例如单击按钮或从列表中选择某项。通过“处理”控件定义的事件，可以改变当用户对某个控件执行相应操作时会发生什么。何时以及如何实现这些事件处理程序，取决于具体的应用程序和具体的控件，但一般来说，对于Button控件，我们都会处理Click事件；对于ListBox控件，则需要用户在用户改变所选项时执行某种操作，因此通常会处理SelectionChanged事件。对于Label、TextBlock等其他控件来说，也许并不需要实现任何事件。

警告： 尽管当我们花一些时间让用户界面变得比标准的Windows界面更有趣时，用户通常都会很乐意接受，但在更改标准的控件行为时，请务必三思。例如，假如我们更改了**Button**控件，使其仅响应用户的右击操作，这会使用户在用鼠标左键单击该按钮之后什么也没有发生，导致他们认为这个应用程序出问题了。实际上，如果仅由于好奇而像这样修改按钮控件，那么使用其他控件类型会比直接去修改**Button**控件的默认行为要好得多。

可通过多种方式将控件添加到窗口中，但最简单的方法是直接将它们从工具箱拖放到设计视图或XAML视图中。

试一试： 将控件添加到窗口中

在学习本章内容的过程中，我们可以自由选择添加控件的方式，可将其从工具箱拖曳到设计视图，也可以手动输入XAML代码。

（1）首先将**Button**控件从工具箱拖曳到设计视图中。请注意观察XAML视图中的代码如何进行相应更新，以反映所做的改变。

（2）现在，拖曳另一个**Button**控件，但这次将其放到XAML视图中，并且放在第一个按钮之后，`</Grid>`标签之前。

示例的说明

在设计视图中看到的结果可能有些令人吃惊——第二个按钮占满整个窗口。将控件拖曳到设计视图时，Visual Studio会尝试自动设置属性并插入其子元素，以便让该控件可以按照标准外观显示。而将同样的控件拖曳到XAML视图时，则不会发生这样的调整，插入的只是用来定义该控件的那个标签而已。

多次尝试将控件放在窗口中希望的特定位置，但发现要将位置放得很准确比较困难。此时，可直接将其拖曳到XAML视图中，或者手动输入相应的代码。

注意： 如果希望拖曳控件，但发现很难将其放到正确位置，可以随意将其放置在某个位置，然后将为其自动生成的相应XAML代码剪切并粘贴到正确位置即可。

14.2.2 属性

如前文所述，所有控件中都包含许多属性，这些属性可用来控制控件的行为。某些属性的含义很容易理解，例如Height和Width，但某些却难以理解，例如RenderTransform。所有属性都可以通过属性（Properties）面板来设置，也可以直接在XAML中设置或直接在设计视图图中进行调整。

注意： 创建一个新项目时，Visual Studio会给类创建一个默认名

称空间。随后在项目中添加新的类或窗口时，就使用该名称空间。在 **Solution Explorer** 中双击 **Properties**，可以更改该名称空间。如果发现类的名称空间不同于示例给出的名称空间，可以把默认名称空间改为本书中的名称空间。这种改变只会影响新类，不影响已在项目中的类。

试一试：控制属性：**Ch14Ex01\MainWindow.xaml**

回到之前的那个示例，并执行下面的操作。在更改属性时，请注意观察这些更改是如何影响XAML和设计视图的。把整个窗口修改为如图14-1所示的样子。

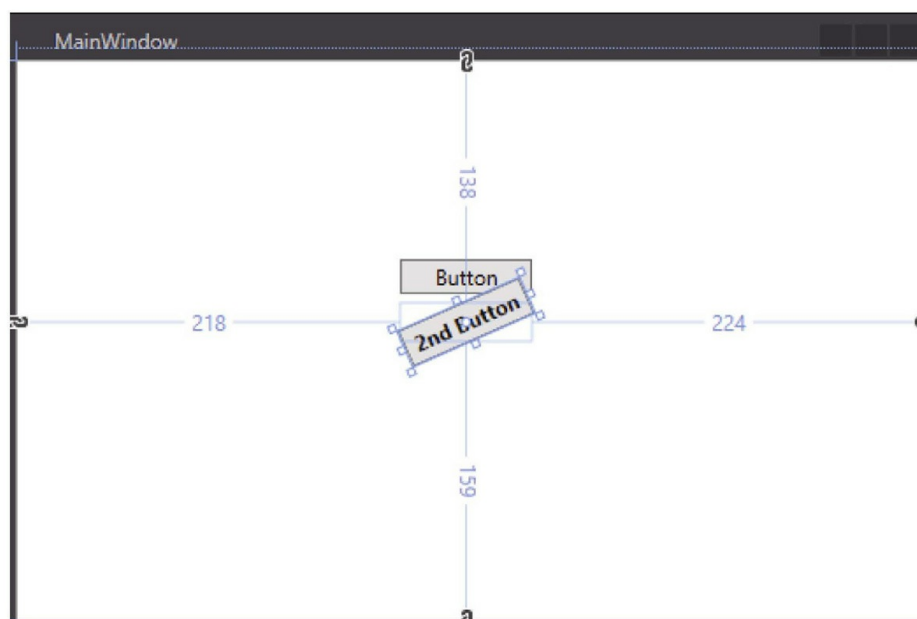


图14-1

(1) 首先在设计视图中选中第二个Button控件；这正是占满整个窗口的那个按钮。

(2) 可在Properties面板的顶部更改该控件名称。将其改为rotatedButton。

(3) 在Common节点下，将Content的值改为2nd Button。

(4) 在Layout下，将Width和Height分别更改为75和22。

(5) 展开Text节点，单击B图标，将文本改为粗体。

(6) 选择第一个Button控件，将其拖曳到第二个Button控件上。Visual Studio会通过贴靠功能帮助调整控件的位置。

(7) 再次选择第二个按钮，将鼠标指针悬停到其左上角。当指针变为带有箭头的四分之一圆弧时，向下拖曳，使该按钮倾斜。

(8) 现在，窗口的XAML代码应该如下所示：

```
<Window x:Class="Ch14Ex01.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2
    xmlns:mc="http://schemas.openxmlformats.org/markup-compa
    xmlns:local="clr-namespace:Ch14Ex01"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
<Grid>
    <Button x:Name="button" Content="Button" HorizontalAlignme
```

```

        Margin="218,113,0,0" VerticalAlignment="Top" Width="100"
<Button x:Name="rotatedButton" Content="2nd Button" Width="100"
        Height="22" FontWeight="Bold" Margin="218,138,224,156"
        RenderTransformOrigin="0.5,0.5" >
<Button.RenderTransform>
    <TransformGroup>
        <ScaleTransform/>
        <SkewTransform/>
        <RotateTransform Angle="-23.896"/>
        <TranslateTransform/>
    </TransformGroup>
</Button.RenderTransform>
</Button>
</Grid>
</Window>

```

(9) 按F5键运行该应用程序，并尝试改变窗口大小。注意，第二个按钮会随着窗口大小的变化而移动，而第一个按钮则保持不动。

示例的说明

在三个视图中，任意一个视图更改都会自动反映在其他视图中，但某些视图可能更适合做某些特定的调整。修改按钮上显示的文本等不重要内容时，可很快在XAML视图中进行修改，但如果要添加一些用于变形渲染的代码，则在设计视图中调整会更快。

在本练习中，我们首先更改了按钮名，该操作为该按钮添加了

x>Name属性。控件的名称必须在整个名称空间范围内是唯一的，也就是说一个名称只能指定给一个控件。

接着更改了**Content**属性，并设置了控件的**Height**和**Width**属性，以及将字体更改为粗体。通过这样的更改，控件在窗口中的外观得到了改变。在之前，该控件占满了整个容器，但现在，我们将其限制为特定大小。

随后将第一个按钮拖曳到设计视图中的特定位置。如本章后面所述，这样的操作并不总是得到相同的结果，而会根据控件所处的容器而有所不同。在本例中，由于有了**Grid**容器，可将控件拖曳到特定位置。拖曳操作会设置控件的**Margin**属性。同时，还需要注意另外两个属性：**HorizontalAlignment="Left"**和**VerticalAlignment="Top"**。设置上述两个属性后，该控件的四周留白将相对于窗口的左上角而定，并且将保留在所放置的那个网格位置中。如果此时对比第一个按钮和第二个按钮，将注意到，第二个控件并未设置上述属性。正是由于没有设置**Alignment**和**Margin**属性，该控件才会停留在容器的中间，即使是在运行时也是这样。也就是说，已经设置了**Alignment**和**Margin**属性的第一个按钮在窗口大小改变时会固定在窗口中的特定位置，而第二个按钮始终保持在窗口中间。

最后一个步骤中做了轻微修改。通过在“旋转”鼠标指针出现的位置拖曳控件，可以旋转该控件。这是**XAML**和**WPF**的一个标准功能，可应用到所有控件中，不过极少数控件在旋转后不会对自己内部的内容做相应调整。这主要是那些依赖**Windows Form**或旧**Windows**控件来显示内容的控件。

在**WPF**中可以进行的变形操作将在第15章中进行介绍，但通过拖曳

鼠标自动产生的XML代码，我们可以看到，只需要控制这些属性，就可以执行一些高级动画效果。

1. 依赖属性

大多数情况下，标准.NET属性都是简单的设置器和获取器，这在大多数时候都是足够使用的。不过，如果要开发能够根据属性的更改来自动或动态做出改变的用户界面，就需要在这些将会重复多次执行的get和set方法中编写许多代码。依赖属性（Dependency Property）是一种能够注册到WPF属性系统中的属性，据此可以获得更多的功能。这些功能包括自动属性更改通知，但此外还有其他很多好处。具体说来，依赖属性的功能包括：

- 可通过样式来更改依赖属性的值。
- 可通过资源或数据绑定来设置依赖属性的值。
- 可在动画中更改依赖属性的值。
- 可按层级结构设置XAML中的依赖属性。也就是说，设置某个父元素中依赖属性的值时，可将该值也作为其子元素中同一个依赖属性的默认值。
- 可通过明确定义的代码模式，来配置属性值更改通知。
- 可配置一系列相关属性，其中一个属性值改变后，会自动更新其他属性。这种功能称为强制（coercion）。这样的操作通常称为被更改的属性强制其他属性的值发生变化。
- 可对依赖属性应用元数据，以便指定其他行为特征。例如，我们可以指定，如果给定的属性值发生变化，就自动调整用户界面。

在实践中，由于依赖属性都通过特定的方法来实现，因此我们可能

不会注意到它们与普通属性有太大的区别。但当我们创建自己的控件时，很快会发现在使用普通.NET属性时，很多功能突然间消失不见了。

第15章将介绍如何实现新的依赖属性。

2. 附加属性

附加属性（Attached Property）是一种在定义该属性的类实例的每个子对象上都可用的属性。例如，如本章稍后所述，在之前的示例中用到的Grid控件可以定义列和行，以便对Grid控件的子控件进行排序。这样，每个子控件就都可以使用Column和Row这两个附加属性来指定自己属于网格中的哪一个单元格了：

```
<Grid HorizontalAlignment="Left" Height="167" VerticalAlignmer
    <Button Content="Button" HorizontalAlignment="Left" Margin=
VerticalAlignment="Top" Width="75" Grid.Column="0" Grid.Row="0

Height="22" />
...
</Grid>
```

在这段代码中，引用附加属性的做法是使用父元素的名称，加上一个句点，后跟附加属性的名称。

在WPF中，附加属性有很多用处。在稍后的14.3节“控件布局”中可以看到许多通过附加属性来指定控件位置的例子。同样，我们也将学习

如何在容器控件中定义附加属性，使子控件可以定义诸如自己要贴靠到容器哪一侧这样的属性。

14.2.3 事件

第13章介绍了什么是事件，以及如何使用它们。本节专门讨论由WPF控件生成的事件，还将介绍一种通常与用户操作关联的路由事件（`routed event`）。例如，当用户单击某个按钮时，该按钮会生成一个事件，用于表明自身发生了什么。通过处理该事件，程序员为该按钮提供了某种功能。

我们要处理的大部分事件都是本书中所涉及控件的通用事件，例如`LostFocus`和`MouseEnter`等。这是因为这些事件本身继承自诸如`Control`或`ContentControl`的基类。此外，像`DatePicker`控件的`CalendarOpened`事件是专用事件，只存在于特定的控件中。表14-1列出了一些最常用的事件。

表14-1 通用控件事件

事件	说明
Click	当控件被单击时发生。某些情况下，当用户按下Enter键时也会发生这样的事件
Drop	当拖曳操作完成时发生，也就是说，当用户将某个对象拖曳到该控件上，然后松开鼠标按钮时发生
DragEnter	当某个对象被拖曳进入该控件的边缘范围内时发生

DragLeave	当某个对象被拖曳出该控件的边缘范围之外时发生
DragOver	当某个对象被拖曳到控件上时发生
KeyDown	当该控件具有焦点，并且某个按键被按下时发生。该事件总在KeyPress和KeyUp事件之前发生
KeyUp	当该控件具有焦点，并且某个按键被释放时发生。该事件总在KeyDown事件后发生
GotFocus	当该控件获得焦点时发生。勿用该事件对控件执行验证操作。应该改用Validating和Validated
LostFocus	当该控件失去焦点时发生。请勿使用该事件对控件执行验证操作。应该改用Validating和Validated
MouseDoubleClick	当双击该控件时发生
MouseDown	当鼠标指针经过某个控件，鼠标按钮被按下时发生。该事件与Click事件并不相同，因为MouseDown事件在按钮被按下后，在其释放前发生
MouseMove	当鼠标经过控件时持续发生
MouseUp	当鼠标指针经过控件，而鼠标按钮又被释放时发生

本章的示例将多次遇到上述这些事件。

1. 处理事件

为事件添加处理程序有两种基本方式。其一是使用Properties窗口中的事件列表，如图14-2所示。当单击Properties窗口中的闪电按钮时，就会出现事件列表。

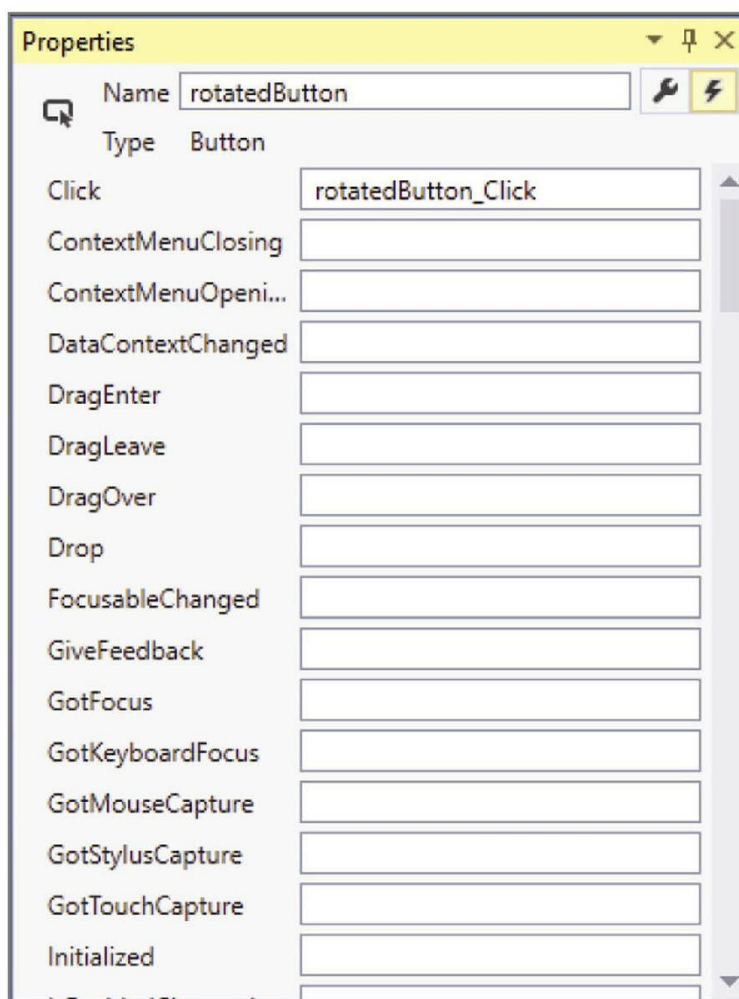


图14-2

要为特定事件添加处理程序，可以输入事件名，然后按回车键，也可以在事件列表中事件名的右侧双击。该操作会将相应事件添加到XAML标签中。而处理该事件的相应方法的签名则会被添加到C#代码隐藏文件中。

```

<Button x:Name="rotatedButton" Content="2nd Button" Width="75"
        Height="22" FontWeight="Bold" Margin="218,138,224,159"
        RenderTransformOrigin="0.5,0.5"
        Click="rotatedButton_Click">
    ...
</Button>

private void rotatedButton_Click(object sender, RoutedEventArgs
    {
    }

```

另外，还可以直接在XAML中输入事件名，并将相应的处理程序名称添加到其中。如果使用这种方法，可右击该事件，然后选择Navigate to Event Handler。这样就能把事件处理程序添加到代码隐藏文件中。

2. 路由事件

WPF中存在一种路由事件（**routed event**）。标准的.NET事件会被显式订阅该事件的代码处理，且只发送到这些订阅者那里。路由事件的不同之处在于，可将事件发送到包含该控件所在层次的所有控件。

当路由事件发生时，它会向发生该事件的控件的上层与下层控件传递。也就是说，如果右击了某个按钮，会首先将**MouseRightButtonDown**事件发送给该按钮本身，然后发送给该控件的父控件，在之前的示例中，就是Grid控件。如果Grid控件未处理该事件，该事件会最终传递给窗口。如果不希望该事件被继续传往更高的控件层次，只需要将**RoutedEventArgs**的属性**Handled**设置为true即可，此时不会再发生其他调用。当某个事件像这样往上层传递时，就称其为冒泡事件（**bobbling**

event)。

路由事件也可以往其他方向传递，例如从根元素传往执行操作的控件。这样的事件被称作下钻事件（tunneling event），并且按照约定，所有这类事件都应该加上Preview前缀，并且总是在相应的冒泡事件之前发生。PreviewMouseButtonDown事件就属于这一类。

最后需要说明的是，路由事件的行为也可以和标准的.NET事件一样，只发送给执行操作的控件。

3. 路由命令

路由命令（routed command）的作用与事件相似，都是引起一些代码开始执行。但事件只能直接与XAML中的单个元素和代码中的一个处理程序绑定，路由命令则更复杂。

事件和命令的关键差异主要在使用过程中体现出来。如果一段代码响应的是只在应用程序中的一个位置发生的用户操作，则应该使用事件。例如，当用户单击某个窗口中的OK按钮以便保存并关闭该窗口时，就使用此类事件。当代码响应多个位置的操作时，则应该使用命令。例如，很多时候，既可以在菜单中选择Save命令，也可以使用某个工具栏按钮来保存应用程序的内容。这样的需求实际上也可以使用事件处理程序来完成，但这意味着我们需要在许多地方编写相同的代码；而使用命令，则只需要编写一次即可。

在创建命令时，还需要通过一些代码来回答这样一个问题：“当前是否允许用户使用这段代码？”也就是说，当将一个命令与某个按钮关联起来时，该按钮可以询问这个命令能否执行，并相应地设置其状态。

实现一个命令比实现一个事件更复杂，所以我们将其放到第15章中，在介绍菜单项时讲解。

试一试：路由事件：**Ch14Ex01\MainWindow.xaml**

下面的示例是在本章之前的示例基础上完成的。如果在之前的练习中添加了行和列，应将它们删除掉，以便符合本示例中XAML代码的要求。

（1）选择rotatedButton按钮，然后添加一个KeyDown事件。可通过Properties面板或直接输入XAML代码的方法来完成这一步骤。将其命名为rotatedButton_KeyDown。

（2）在XAML视图中单击Grid对应的标签将其选中，然后为其添加相同事件。将其命名为Grid_KeyDown。

（3）在XAML视图中选择Window标签，再次添加该事件。将其命名为Window_KeyDown。

（4）重复步骤（1）到（3），所不同的是添加PreviewKeyDown事件，随后修改事件的名称，表明它们是Preview处理程序。最终的XAML代码如下所示：

```
<Window x:Class="Ch14Ex01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend
```

```

xmlns:mc="http://schemas.openxmlformats.org/markup-com
xmlns:local="clr-namespace:Ch14Ex01"
mc:Ignorable="d"
Title="MainWindow" Height="350" Width="525" KeyDown="W
PreviewKeyDown="Window_PreviewKeyDown">
    <Grid KeyDown="Grid_KeyDown" PreviewKeyDown="Grid_PreviewKe
x:Name="button" Content="Button" HorizontalAlignment="Left"
    Margin="27,4,0,0" VerticalAlignment="Top" Width="75" G
    Grid.Row="0"/>
    <Button x:Name="rotatedButton" Content="2nd Button" Width="75"
    FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
    KeyDown="rotatedButton_KeyDown"
    PreviewKeyDown="rotatedButton_PreviewKeyDown" Grid.Co
    Grid.Row="1" >
    <Button.RenderTransform>
        <TransformGroup>
            <ScaleTransform/>
            <SkewTransform/>
            <RotateTransform Angle="-23.896"/>
            <TranslateTransform/>
        </TransformGroup>
    </Button.RenderTransform>
</Button>
</Grid>
</Window>

```

(5) 如果直接输入XAML代码，则右击每个事件，通过选择

Navigate to Event Handler菜单项在代码隐藏文件中添加事件处理程序。

(6) 将下列代码添加到事件处理程序：

```
private void Grid_KeyDown(object sender, KeyEventArgs e)
{
    MessageBox.Show("Grid handler, bubbling up");
}
private void Grid_PreviewKeyDown(object sender, KeyEventArgs
{
    MessageBox.Show("Grid handler, tunneling down");
}
private void rotatedButton_KeyDown(object sender, KeyEventArg
{
    MessageBox.Show("rotatedButton handler, bubbling up");
}
private void rotatedButton_PreviewKeyDown(object sender, KeyE
{
    MessageBox.Show("rotatedButton handler, tunneling down");
}
private void Window_KeyDown(object sender, KeyEventArgs e)
{
    MessageBox.Show("Window handler, bubbling up");
}
private void Window_PreviewKeyDown(object sender, KeyEventArg
{
    MessageBox.Show("Window handler, tunneling down");
```

```
}
```

(7) 按下F5键运行该程序。

(8) 通过单击并按下任意键（回车键、Tab键、Esc键、空格键和方向键除外）的方式选择旋转后的按钮。观察一下事件的执行顺序。

(9) 关闭该应用程序。

(10) 找到Grid_PreviewKeyDown事件处理程序，在MessageBox一行的下方添加如下代码：

```
e.Handled = true;
```

(11) 重复步骤（7）和（8）。

示例的说明

KeyDown和PreviewKeyDown事件演示了冒泡事件和下钻事件。在选择rotatedButton按钮的同时按下某个键时，会看到每个事件处理程序被依次执行。

首先执行Preview事件，从Window对象的处理程序开始，然后是Grid，最后是rotatedButton。随后执行KeyDown事件，这次的执行顺序与上面正好相反，从rotatedButton按钮的事件处理程序开始，到Window对象的处理程序结束。

如果在步骤（8）中按下的是明确说明不要按的那几个键，会发现Preview事件将只会在Window对象上执行。这是因为上述几个按键并不

被看成输入按键，它们会被Grid和Button忽略掉。

随后添加了这行代码：

```
e.Handled = true;
```

该代码戏剧性地改变了程序的执行方式。设置RoutedEventArgs的Handled属性不仅会执行下钻事件，也会执行冒泡事件。对于所有此类事件来说，基本上都是这样的。如果终止执行Preview或“普通”事件处理程序中的一种，两者都会终止。

4. 控件类型

如前所述，在WPF中有很多控件可供使用。它们分为内容控件和项控件两大类。内容控件（例如Button控件）有一个Content属性，可将这个属性设置为其他任意的控件。也就是说，可以决定控件的显示方式，但只能在内容中直接放置一个控件。对于项控件来说，可以在其中插入多个控件作为其内容。Grid控件就是项控件的一个典型例子。在创建用户界面时，会将这两种控件混合起来使用。

除内容控件和项控件外，还有其他一些类型的控件不允许在其中放置控件作为它们的内容。Image控件就属于这种情况，该控件只能用来显示图片。更改控件的行为会改变控件的作用。

14.3 控件布局

到这里为止，本章使用Grid元素来设计一些控件的布局，这主要是因为在新建一个WPF应用程序时，它是默认的布局控件。不过，我们还没有介绍这一控件的所有功能，也没有介绍除此之外其他能用来进行布局的容器。本节将进一步介绍控件布局，这是WPF的一项基本概念。

所有内容布局控件都继承自抽象的Panel类。该类仅定义了一个容器，可以容纳继承自UIElement的对象集合。实际上，所有WPF控件都继承自UIElement。我们不能直接使用Panel类对控件进行布局，但可以从它派生出其他需要的控件。可直接使用以下这些继承自Panel的布局控件：

- **Canvas**——该控件允许以任何合适的方式放置子控件。它不会对子控件的位置施加任何限制，但不会对位置摆放提供任何辅助。
- **DockPanel**——该控件可让其中的子控件贴靠到自己四条边中的任意一边。最后一个子控件则可以充满剩余区域。
- **Grid**——该控件让子控件的定位变得比较灵活。可将该控件的布局分为若干行和若干列，这样就可以在网格布局中对齐控件。
- **StackPanel**——该控件能够按照水平方向或垂直方向依次对子控件进行排列。
- **WrapPanel**——与StackPanel一样，该控件也能按照水平方向或垂直方向依次对子控件进行排列，但它不是按照一行或一列来排序，而是根据可用空间大小以多行多列的方式来排列。

稍后就将详细介绍如何使用这些控件。但首先需要了解几个基本概

念：

- 控件如何以堆叠顺序排列
- 如何使用对齐、边距和填充来定位控件及其内容
- 如何使用Border控件

14.3.1 堆叠顺序

当某个容器控件包含多个子控件时，这些子控件会按特定的堆叠顺序进行排列。如果使用过绘图软件，可能已经熟悉了这个概念。我们可以将堆叠顺序想象为，每个控件都包含在一个玻璃盘中，而容器包含一摞这样的玻璃盘。这样一来，容器的外观看起来就类似于从这些玻璃盘的上方往下看时的样子。当容器中的控件重叠时，我们看到的最终结果就由这些玻璃盘的上下堆叠顺序来决定。如果某个控件位于上层，在重叠的部分，该控件就是可见的。而下层的控件则可能会被它们上层的控件遮挡住一部分或全部。

堆叠顺序也影响在窗口中进行鼠标单击时的点中行为。如果考虑控件的上下堆叠情况，被点中的控件则总是在最上层的那一个。而控件的堆叠顺序则是由这些控件在容器的子控件列表中出现的顺序来决定的。容器中的第一个子控件位于最下方，而最后一个子控件则位于最上方。在这两者之间的子控件则按照出现的顺序自下自上排列。此外，控件的堆叠顺序还会对在WPF中使用的某些布局控件产生其他影响，稍后将介绍相关内容。

14.3.2 对齐、边距、填充和尺寸

前面的示例中用到了Margin、HorizontalAlignment和VerticalAlignment属性在Grid容器中安排控件的位置，但当时并没有对它们进行详细介绍。另外，我们了解了如何使用Height和Width来指定控件的维度。上述这些属性，以及尚未介绍过的Padding属性一起，在大多数甚至所有布局控件（稍后将看到）中都十分有用，只不过它们各自的作用有所不同。不同的布局控件也可对这些属性设置一些默认值。接下来将会看到许多相关的例子，不过首先了解与此相关的基本知识。

HorizontalAlignment和VerticalAlignment这两个对齐属性确定控件的对齐方式。可将HorizontalAlignment设置为Left、Right、Center或Stretch。Left和Right用于让控件对齐容器的左边缘或右边缘，Center则表示位于中间，Stretch则自动调整控件宽度，使其接触到容器的左右边缘。VerticalAlignment与此类似，但值为Top、Bottom、Center或Stretch。

Margin和Padding分别用于指定控件边缘外侧和内侧的留白。之前的示例使用Margin属性让控件与窗口的边缘保持一定距离。由于还将HorizontalAlignment设置为Left，VerticalAlignment设置为Top，因此控件会保持在左上角特定的位置上，Margin属性使其与容器边缘保持了一定的距离。Padding与此类似，所不同的只是它用来指定控件内容与控件边缘的距离。这对于指定控件的Border比较有用，下一节将介绍Border控件。Padding和Margin可按照四个方向来指定（形式为leftAmount、topAmount、rightAmount、bottomAmount），也可以指定一个值（Thickness值）。

稍后，你将了解到Height和Width属性往往被其他属性所控制。例如，当HorizontalAlignment设置为Stretch时，控件的Width属性就会随着容器宽度的改变而改变。

14.3.3 Border控件

Border控件是一种非常简单，却非常有用的容器控件。它内含一个子对象，而不像稍后将介绍的其他复杂控件那样内含多个子对象。该子对象会完全充满整个Border控件。这看起来不是特别有用，但请记住，可使用Margin和Padding属性来指定Border在其容器中的位置，以及Border中的内容相对于Border本身的位置。还可以为Border设置诸如Background这样的属性，使其可见。接下来将实际使用这一控件。

14.3.4 Canvas控件

之前曾经提到过，Canvas控件可以完全自由地对控件的位置进行安排。同时，对Canvas的子元素应用HorizontalAlignment和VerticalAlignment属性并不能改变这些元素的位置。

如之前的例子所示，可使用Margin属性来定位元素，但最好使用Canvas类公开的Canvas.Left、Canvas.Top、Canvas.Right和Canvas.Bottom附加属性。

```
<Canvas...>  
    <Button Canvas.Top="10" Canvas.Left="10"...>Button1</Button  
</Canvas>
```

上面这段代码将Button控件定位到距离Canvas控件顶部和左侧各10像素的位置。需要注意，Top和Left属性的优先级高于Bottom和Right属性。例如，如果同时指定Top和Bottom属性，Bottom属性会被忽略掉。

图14-3分别展示了在Canvas控件中放置两个Rectangle控件，并将窗口调整为两种不同大小之后的情况。

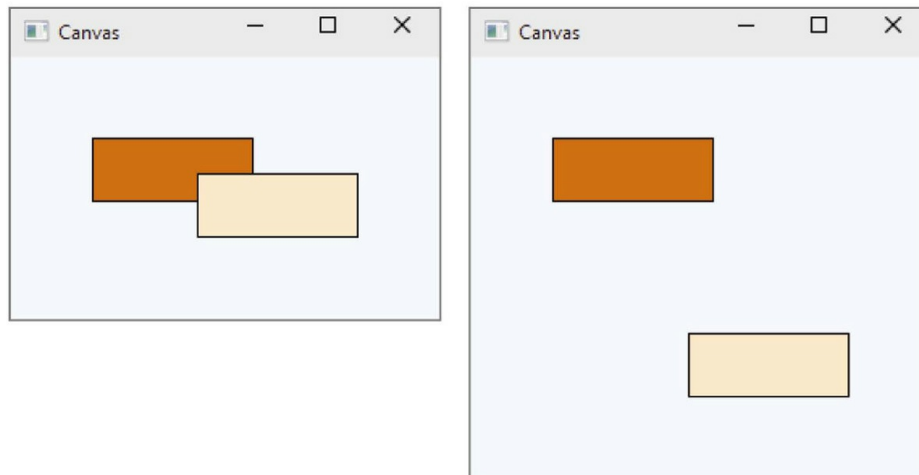


图14-3

注意： 本节中所有示例布局都可以在本章对应的下载代码的LayoutExamples项目中找到。

其中一个Rectangle控件的位置相对于左上角进行设置，而另一个则相对于右下角进行设置。调整窗口大小时，它们各自的相对位置保持不变。还可以看到Rectangle控件堆叠顺序的重要性。右下角的Rectangle控件位于上层，所以当两者重叠时，用户看到的是这个控件。

本示例的代码如下所示（可在LayoutExamples\Canvas.xaml下载文件中找到）：

```
<Window x:Class="LayoutExamples.Canvas"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/preser
```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:LayoutExamples"
mc:Ignorable="d"
Title="Canvas" Height="300" Width="300">
<Canvas Background="AliceBlue">
    <Rectangle Canvas.Left="50" Canvas.Top="50" Height="40" Width="40"
    Stroke="Black" Fill="Chocolate" />
    <Rectangle Canvas.Right="50" Canvas.Bottom="50" Height="40" Width="40"
    Stroke="Black" Fill="Bisque" />
</Canvas>
</Window>

```

14.3.5 DockPanel控件

顾名思义，DockPanel控件允许将控件贴靠到某条边上。就算之前我们没有特别注意过这样的布局方式，也应该十分熟悉此类布局。例如，Word软件中的功能区（Ribbon）控件就停留在Word窗口顶部，VS中的各种窗口也各自停靠在位置上。并且，可以拖动VS中的这些窗口，改变它们的停靠位置。

DockPanel具有一个能让子控件用来指定停靠边缘的附加属性，即DockPanel.Dock。可将该属性的值设置为Left、Top、Right或Bottom。

DockPanel中控件的堆叠顺序非常重要，因为每当一个控件停靠到某个边缘上后，其他子控件的可占用空间就会减少。例如，将一个工具

栏控件停靠到DockPanel的顶部，然后将另一个工具栏停靠到DockPanel的左边。这样一来，第一个控件就会占满DockPanel显示区域的整个顶部，而第二个控件则只能占满第一个控件的底部到DockPanel控件底部的左侧区域。

最后一个子控件通常将只能占满其他子控件之外余下部分的相应区域（可控制这一行为，所以前面这句话并不是完全肯定的语气）。

在DockPanel中定位一个控件时，该控件所占用的区域可能会小于DockPanel为其保留的区域。例如，如果将一个宽度为100、高度为50、HorizontalAlingment的值为Left的Button控件停靠到DockPanel的顶部，在Button的右侧就会留下一部分无法被其他停靠子控件占用的区域。并且，如果Button控件的Margin值为20，DockPanel顶部被保留的区域就有90像素高（控件的高度与上下两边的Margin值相加）。在使用DockPanel设置布局时，务必考虑这些因素；否则可能无法获得预想的结果。

图14-4展示了一个DockPanel布局示例。

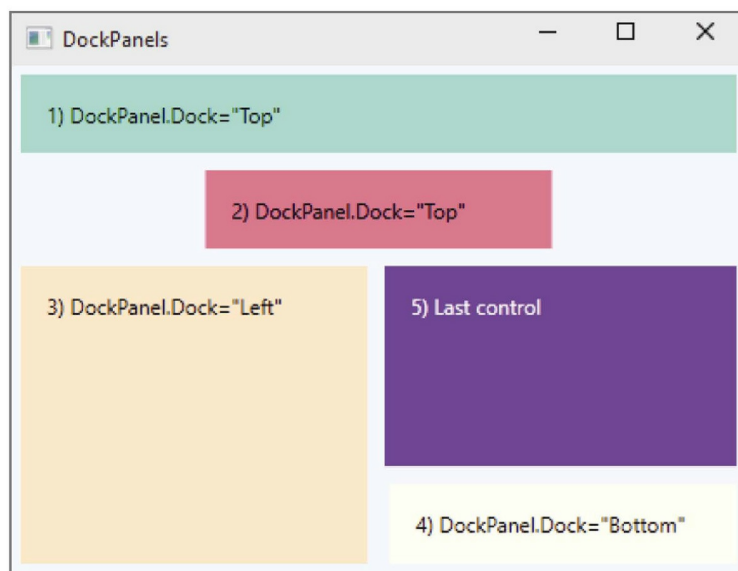


图14-4

这一布局的代码如下所示（可在LayoutExamples\DockPanel.xaml下载文件中找到）：

```
<Window x:Class="LayoutExamples.DockPanels"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:LayoutExamples"
    mc:Ignorable="d"
    Title="DockPanels" Height="300" Width="300">
    <DockPanel Background="AliceBlue">
        <Border DockPanel.Dock="Top" Padding="10" Margin="5"
            Background="Aquamarine" Height="45">
            <Label>1) DockPanel.Dock="Top"</Label>
        </Border>
        <Border DockPanel.Dock="Top" Padding="10" Margin="5"
            Background="PaleVioletRed" Height="45" Width="200">
            <Label>2) DockPanel.Dock="Top"</Label>
        </Border>
        <Border DockPanel.Dock="Left" Padding="10" Margin="5"
            Background="Bisque" Width="200">
            <Label>3) DockPanel.Dock="Left"</Label>
        </Border>
        <Border DockPanel.Dock="Bottom" Padding="10" Margin="5"
```

```
Background="Ivory" Width="200" HorizontalAlignment="Right">
    <Label>4) DockPanel.Dock="Bottom"</Label>
</Border>

<Border Padding="10" Margin="5" Background="BlueViolet">
    <Label Foreground="White">5) Last control</Label>
</Border>

</DockPanel>

</Window>
```

上述代码使用前面介绍的Border控件标记出示例布局中停靠控件的区域，并使用Label控件输出一些简单的描述性文字。要了解整个布局，必须从头到尾阅读这段代码，分别来看看每个控件的情况：

（1）第1个Border控件停靠于DockPanel控件的顶部。DockPanel中被占去的区域为顶部的55像素（Height加上两个Margin）。需要注意，Padding属性不影响这一布局，因为该属性只会应用到Border的内部，并不能控制嵌入的Label控件的位置。如果未指定Height或Width属性，Border控件会占满其所停靠边缘的整个可用区域，这就是为什么它会横跨整个DockPanel控件的原因。

（2）第2个Border控件同样停靠到DockPanel的顶部，并占用了剩下部分顶部的55像素高的区域。该Border控件还有一个Width属性，这就使其仅占用了DockPanel一部分的宽度。DockPanel中HorizontalAlignment属性的默认值为Center，所以它位于DockPanel的中间。

（3）第3个Border控件停靠到DockPanel的左侧，占用了左侧210像素的区域。

（4）第4个Border控件停靠在DockPanel底部，占用的区域为30像素加上其中Label控件（也可以是其他控件）的高度。该高度由Margin、Padding和Border控件的内容共同决定，并没有明确指定。Border控件被固定到DockPanel的右下角，因为其HorizontalAlignment值为Right。

（5）第5个（也就是最后一个Border控件）占满了其他所有区域。

运行该示例，然后试着调整内容的大小。注意，控件在堆叠顺序中越接近顶层，就越具有占用空间的优先权。在缩小窗口时，第5个Border控件可能会被上层的其他所有控件完全遮盖。所以注意在使用DockPanel控件进行布局时避免这种情况的发生，例如可为窗口设置允许的最小尺寸。

14.3.6 StackPanel控件

可将StackPanel控件理解为精简版的DockPanel，即子控件所停靠的边缘是固定不变的。另一个差异是StackPanel中的最后一个子控件不会占满剩余空间。不过，这些子控件默认情况下会拉伸到StackPanel控件的边缘。

控件的堆叠方向由三个属性决定。Orientation可设置为Horizontal或Vertical，HorizontalAlignment和VerticalAlignment可用于决定控件的堆栈是紧靠StackPanel的顶部、底部、左侧还是右侧进行排列。还可将对齐（Alignment）属性的值设置为Center，让控件在StackPanel的中间堆叠。

图14-5展示了两个StackPanel控件，其中分别包含三个按钮。上方

的StackPanel控件的Orientation属性设置为Horizontal，下方的StackPanel控件的Orientation属性则设置为Vertical。

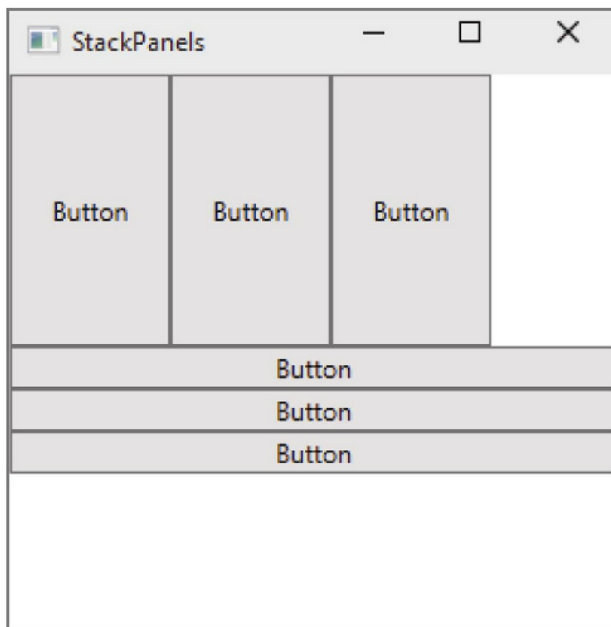


图14-5

此处所用到的代码如下所示（可在LayoutExamples\StackPanels.xaml下载文件中找到）：

```
<Window x:Class="LayoutExamples.StackPanels"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:LayoutExamples"
    mc:Ignorable="d"
    Title="StackPanels" Height="300" Width="300">
    <Grid>
```



```

        <StackPanel HorizontalAlignment="Left" Height="128" Vert
Width="284" Orientation="Horizontal">
            <Button Content="Button" Height="128" VerticalAlignmen
                Width="75"/>
            <Button Content="Button" Height="128" VerticalAlignment
                Width="75"/>
            <Button Content="Button" Height="128" VerticalAlignmen
                Width="75"/>
        </StackPanel>
        <StackPanel HorizontalAlignment="Left" Height="128" Verti
Width="284" Margin="0,128,0,0" Orientation="Vertical">
            <Button Content="Button" HorizontalAlignment="Left" Wid
            <Button Content="Button" HorizontalAlignment="Left" Wid
            <Button Content="Button" HorizontalAlignment="Left" Wid
        </StackPanel>
    </Grid>
</Window>

```

14.3.7 WrapPanel控件

WrapPanel基本上可以认为是StackPanel的扩展版本；容纳不下的控件会被安排到下一行（或下一列）。图14-6展示了一个包含多个形状的WrapPanel控件，其窗口被调整为两种不同大小。

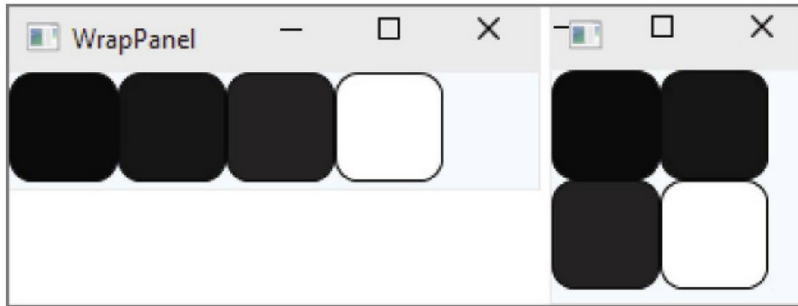


图14-6

实现该效果的代码如下所示（可在LayoutExamples\WrapPanel.xaml 下载文件中找到）：

```
<Window x:Class="LayoutExamples.WrapPanel"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:LayoutExamples"
    mc:Ignorable="d"
    Title="WrapPanel" Height="92" Width="260">
    <WrapPanel Background="AliceBlue">
        <Rectangle Fill="#FF000000" Height="50" Width="50" Stroke="Black"
            RadiusX="10" RadiusY="10" />
        <Rectangle Fill="#FF111111" Height="50" Width="50" Stroke="Black"
            RadiusX="10" RadiusY="10" />
        <Rectangle Fill="#FF222222" Height="50" Width="50" Stroke="Black"
            RadiusX="10" RadiusY="10" />
        <Rectangle Fill="#FFFFFFFF" Height="50" Width="50" Stroke="Black"
            RadiusX="10" RadiusY="10" />
    </WrapPanel>
</Window>
```

```
        RadiusX="10" RadiusY="10" />
    </WrapPanel>
</Window>
```

WrapPanel控件是创建动态布局的好方法，使用户可以精确地控制内容的显示。

14.3.8 Grid控件

Grid控件可以分为多行和多列，以便摆放子控件。本章已经多次提到Grid控件了，但每次都只使用一行和一列而已。要添加更多行和列，可使用RowDefinitions和ColumnDefinitions属性，这两个属性分别是RowDefinition和ColumnDefinition对象的集合，而且是通过属性元素语法来指定的：

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    ...
</Grid>
```

上述代码定义了一个包含两行和两列的Grid控件。注意，这里并不需要其他信息；每一行和每一列都会随着Grid控件大小的改变而自动改变大小。每一行占用Grid中三分之一的高度，每一列则占用其一半的宽度。通过将Grid.ShowGridlines属性设置为true，可让Grid控件显示单元格之间的分界线。

注意： 也可以通过在设计视图中单击网格的边缘来定义行和列。当鼠标指针移到网格边缘时，设计视图上会出现一条横穿的黄线；如果单击这条边，就可以插入所需的XAML代码。这样操作后，行和列的Width和Height属性会由设计器自动设定，但我们可以删除这两个属性，或者拖曳相应的线条，以满足我们的需要。

可通过Width、Height、MinWidth、MaxWidth、MinHeight和MaxHeight属性来重新调整大小。例如，为某一列设置Width属性可以使其保持在该宽度。也可将列的Width属性设置为*，这表示“在计算其他所有列的宽度后，占满剩余的空间。”这个值实际上就是默认值。如果有多列的Width为*，这些列会平均占据可用的剩余空间。行的Height属性也可以使用*这个值。Height和Width还可以取值为Auto，也就是根据行和列中的内容来确定自身的高度和宽度。还可以使用GridSplitter控件让用户可以通过鼠标单击并拖曳的方式自行调整行和列的大小。

Grid控件的子控件可使用Grid.Column和Grid.Row附加属性来指定自己属于哪个单元格。这两个属性的默认值都是0，也就是说，如果不填写该属性，子控件会默认位于左上角的单元格中。子控件还可以使用Grid.ColumnSpan和Grid.RowSpan属性来使自己横跨表格中的多个单元

格，其左上角的单元格由Grid.Column和Grid.Row属性指定。

试一试：使用行和列：**Ch14Ex01\MainWindow.xaml**

现在回头看看本章开头介绍的包含两个按钮的那个示例，然后执行以下步骤：

（1）在XAML视图中单击选中Grid控件。

（2）在设计视图中将鼠标指针移动到网格顶部；将会看到一条橙色的线条贯穿整个网格。留下一个按钮的空间，然后单击该线条，生成两列。

（3）在窗口左侧重复步骤（2），生成两行。

（4）选择两个按钮中的第一个。注意，添加行和列的操作实际上已经自动为该按钮添加了Grid.Row和Grid.Column属性。将这两个附加属性的值设置为0。

（5）对Margin属性进行必要的调整，让按钮在单元格中完全可见。

（6）第二个按钮也已改变了，例如添加了Margin值。现在，将第二个按钮的Margin属性删除掉。

（7）在XAML视图添加一个GridSplitter控件，将其放在Grid控件结束标签的上一行，并按照如下方式设置其属性：

```
<GridSplitter Grid.RowSpan="2" Width="3" BorderThickness="2" E
```

(8) 运行该应用程序。完整的XAML代码如下所示：

```
<Window x:Class="Ch14Ex01.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2
    xmlns:mc="http://schemas.openxmlformats.org/markup-compa
    xmlns:local="clr-namespace:Ch14Ex01"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525" KeyDown="Wir
    PreviewKeyDown="Window_PreviewKeyDown">
<Grid KeyDown="Grid_KeyDown" PreviewKeyDown="Grid_PreviewKeyDc
    <Grid.RowDefinitions>
        <RowDefinition Height="109*"/>
        <RowDefinition Height="210*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="191*"/>
        <ColumnDefinition Width="326*"/>
    </Grid.ColumnDefinitions>
    <Button x:Name="button" Content="Button" HorizontalAlignmen
        Margin="27,4,0,0" VerticalAlignment="Top" Width="75" G
        Grid.Row="0"/>
    <Button x:Name="rotatedButton" Content="2nd Button" Width="
        FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
```

```

        KeyDown="rotatedButton_KeyDown"
        PreviewKeyDown="rotatedButton_PreviewKeyDown" Grid.Col
        Grid.Row="1" >
<Button.RenderTransform>
    <TransformGroup>
        <ScaleTransform/>
        <SkewTransform/>
        <RotateTransform Angle="-23.896"/>
        <TranslateTransform/>
    </TransformGroup>
</Button.RenderTransform>
</Button>
<GridSplitter Grid.RowSpan="2" Width="3" BorderThickness="2
        BorderBrush="Black" />
</Grid>
</Window>

```

如图14-7所示，在应用程序运行时，分隔栏被拉到不同位置上。

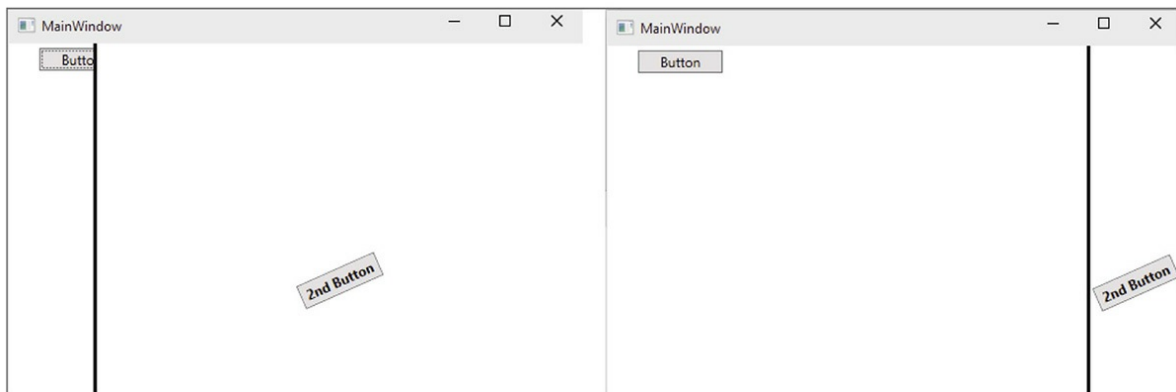


图14-7

示例的说明

通过将网格控件分隔为两列和两行，我们对子控件在网格中的定位方式进行了修改。将第一个按钮的Grid.Row和Grid.Column属性设置为0后，就将其从原位置移到左上角。

第二个按钮看起来并没有怎么变化，但当拖曳GridSplitter控件的分隔线时，可以看到该按钮的边距现在是相对于所在列的左边缘而言的，也就是说在窗口中左右移动分隔线时，按钮也会随之在窗口中左右移动。

14.4 游戏客户端

现在，我们已经了解了WPF和Visual Studio的基本使用方法，接下来就使用控件创建实际应用。本章剩余部分和第15章将主要介绍如何为之前章节中所开发的纸牌游戏编写一个游戏客户端。将用到很多控件来编写这个游戏客户端，随后，你也可以完全自己编写一个。

本章将为这个游戏写一些支持性的对话框——包括About、Options和New Game窗口。

14.4.1 About窗口

About窗口有时称为About对话框，用于显示开发人员及应用程序本身的一些信息。某些About窗口十分复杂，例如Microsoft Office和Visual Studio中的About窗口还显示了版本和许可信息。在应用程序中，Help菜单的最后一个菜单项通常用来打开About窗口。

接下来要创建的这个对话框如图14-8所示。



图14-8

1. 设计用户界面

用户并不会频繁地使用About窗口。实际上，之所以把它放在Help菜单中，是因为只有当用户需要查看应用程序版本信息，或者当应用程序出问题后需要寻找联系方式的时候才会访问它。但这也意味着它对用户是有用的，所以既然要在应用程序中设计这个窗口，就需要重视它。

设计一个应用程序时，应当尽量保持外观和风格的一致性。也就是说，应当在整个应用程序中使用几种固定的颜色，并在不同位置使用相同的控件样式。在Karli Cards这个游戏中，我们将主要使用三种颜色——红色、黑色和白色。

如果观察图14-8，会发现该窗口左上角是Wrox出版社的徽标。之前还没有使用过图像，但在应用程序中添加一些特定的图像可以让用户界面看起来更专业。

2. Image控件

Image是一个简单却效果非凡的控件。它可以显示一幅图片，并按照需要对其进行适当的大小调整。该控件公开了两个属性，如表14-2所示。

表14-2 Image控件

属性	说明
Source	该属性用于指定图像位置。既可以是磁盘上的某个位置，也可以是Web上的某个位置。如第15章所述，也可以创建一个静态资源，并将其作为图像源使用
Stretch	实际上，图片大小很少正好符合我们的需要，并且很多时候图片的大小还需要随着应用程序窗口的改变而改变。可使用该属性来控制图像如何进行大小调整。可用值包括： None——永远不会调整图像大小。 Fill——拉伸图片，使其充满整个可用区域。这可能改变图片的比例。 Uniform——保持图片的宽高比，如果改变了宽高比，不会充满所有可用区域。 UniformFill——在保持宽高比的同时充满整个可用区域。如果在保持宽高比的情况下图片大于可用区域，就会裁减掉超出范围的区域，以适应可用区域的大小

3. Label控件

在之前的示例中，已经见过此类最简单的控件了。它向用户显示简单的文本信息，某些情况下还显示相关的快捷键。它使用Content属性来显示文本信息，使用Label控件显示单行文字。如果在某个字母前加上

下划线“_”前缀，那么该字母在控件中显示时会带有下划线，并且通过Alt与该字母的组合就可以直接访问该控件。例如，_Name可以为这个Label所在的控件直接指定Alt+N快捷键。

4. TextBlock控件

与Label控件类似，该控件也用于显示不含任何复杂格式信息的简单文本。但与Label不同的是，TextBlock控件可以显示多行文字。不能对其组成文字单独设置格式。

TextBlock直接显示文字内容，即使所在控件没有足够的空间来显示文字内容也是这样。当内容过多时，该控件并不会显示滚动条，但可在需要时将其放在一个简单的视图控件ScrollView中，来解决这一问题。

5. Button控件

与Label控件一样，之前也介绍过Button控件。它可用在用户界面的任何地方，而且易于识别。用户可以单击这个控件来完成某种操作——但也仅此而已。如果试图改变其功能，往往导致糟糕的界面设计，让用户感到困惑不解。

默认情况下，Button上可显示一行简短文本或一幅图片，来介绍单击该控件之后所执行的操作。

Button控件并不包含任何用于显示图片或文本的属性，但可使用Content属性来显示简单文本，或在Content中嵌入一个Image控件来显示

图片。相关代码可在Ch14Ex01\ImageButton.xaml下载文件中找到：

```
<Button HorizontalAlignment="Left" VerticalAlignment="Top" Width="160" Height="40">
    <StackPanel Orientation="Horizontal">
        <Image Source=".\\Images\\Delete_black_32x32.png" Stretch="Fill" Width="16" Height="16" />
        <TextBlock>Delete</TextBlock>
    </StackPanel>
</Button>
```

注意： 上述按钮中用到的图片位于下载代码的Ch14Ex01\Images文件夹中。

图14-9展示了一个同时包含文字和图像的Delete按钮。



图14-9

注意： 要完成下面这个示例，需要一个用作横幅的图像文件。该文件所在位置为KarliCards Gui\Images\Banner.png。



试一试：创建About窗口：KarliCards Gui\About.xaml

在开始创建About窗口前，需要新建一个项目。除本窗口外，本章和下一章还会创建好几个窗口，因此，请新建一个WPF应用程序项目，并将其命名为KarliCards GUI。将相应的解决方案命名为KarliCards。

(1) 在Solution Explorer中右击KarliCards Gui项目，然后选择Add|Window，并将该窗口命名为About.xaml。

(2) 通过单击并拖曳，或通过设置其属性的方式来调整窗口大小：

```
Height="300" Width="434" MinWidth="434" MinHeight="300"
ResizeMode="CanResizeWithGrip"
```

(3) 选择Grid控件，然后单击网格边缘来创建4行。不必考虑每一行的准确位置，只需要将它们的值改为如下所示即可：

```
<Grid.RowDefinitions>
    <RowDefinition Height="58"/>
    <RowDefinition Height="20"/>
    <RowDefinition />
    <RowDefinition Height="42"/>
</Grid.RowDefinitions>
```

(4) 将工具箱中的Canvas控件拖曳到最上面一行。删掉由Visual Studio插入的所有属性，然后添加以下代码：

```
Grid.Row="0" Background="#C40D42"
```

(5) 选中新建的Canvas控件，将一个Image控件拖曳到其中。修改其属性，如下所示：

```
Height="56" Canvas.Left="0" Canvas.Top="0" Stretch="UniformToFill"
Source=".\\Images\\Banner.png"
```

(6) 右击该项目，然后选择Add|New Folder。将新建的这个文件夹命名为Images。

(7) 在Solution Explorer中右击新建的文件夹，选择Add|Existing Item。浏览到本章用到的图片。选中所有这些图片，然后单击Add。这样，横幅就会显示在设计视图中。

(8) 选中Canvas控件，并将一个Label控件拖曳到其中。修改其属性，如下所示：

```
Canvas.Right="10" Canvas.Top="25" Content="Karli Cards"
Foreground="#FFF7E7EF" FontFamily="Times New Roman"
```

(9) 选中Grid控件，并将一个新的Canvas控件拖曳到其中。将其属性修改为：

```
Grid.Row="1" Background="Black"
```

(10) 选中新建的Canvas控件，并将一个Label控件拖曳到其中。将其属性修改为：

```
Canvas.Left="5" Canvas.Top="0" FontWeight="Bold" FontFamily="A
```

```
Foreground="White"
```

```
Content="Karli Cards (c) Copyright 2012 by Wrox Press and all
```

(11) 再次选中Grid控件，将最后一个Canvas控件拖曳到最下的一行中。将其属性修改为：

```
Grid.Row="3"
```

(12) 选中新建的Canvas控件，将一个Button控件拖曳到其中。将其属性修改为：

```
Content="_OK" Canvas.Right="12" Canvas.Bottom="10" Width="75"
```

(13) 再次选中Grid控件，将一个StackPanel控件拖曳到最后一个居中的行。将其属性修改为：

```
Grid.Row="2"
```

(14) 选中该StackPanel控件，依次将两个Label控件和一个TextBlock控件拖曳到其中。

(15) 用如下代码修改最上方的Label控件：

```
Content="CardLib and Idea developed by Karli Watson" HorizontalAlignment="Left" VerticalAlignment="Top" Padding="20,20,0,0" FontWeight="Bold" Foreground="#FF8B6F6F"
```

(16) 修改接下来的Label控件，如下所示：

```
Content="Graphical User Interface developed by Jacob Hammer" HorizontalAlignment="Left" Padding="20, 0,0,0" VerticalAlignme
```



```
FontWeight="Bold" Foreground="#FF8B6F6F"
```

(17) 修改TextBlock，如下所示：

```
Text="Karli Cards developed with Visual C# 6 for Wrox Press.  
You can visit Wrox Press at http://www.wrox.com."  
Margin="0, 10,0,0" Padding="20,0,0,0" TextWrapping="Wrap"  
HorizontalAlignment="Left" VerticalAlignment="Top" Height="39"
```

(18) 双击该按钮，在事件处理程序中添加如下代码：

```
private void Button_Click(object sender, RoutedEventArgs e)  
{  
    this.Close();  
}
```

(19) 在Solution Explorer中双击App.xaml文件，将文件名MainWindow.xaml改为About.xaml。

(20) 运行该应用程序。

示例的说明

开始时在该窗口中设置了一些属性。通过设置MinWidth和MinHeight属性，可以防止用户将窗口大小调整到遮挡住内容的程度。ResizeMode属性被设置为CanResizeWithGrip，这可以让窗口的右下角出现一个小手柄标志，让用户知道该窗口的大小是可以调整的。

接下来为网格添加4行。为此，定义窗口的基本结构。通过将第1、

2和4行设置为固定高度，确保只有第3行的高度是可变的；这是包含内容的那一行。

随后添加了第一个Canvas控件。该控件可以轻松地设置第一行的背景色。通过确保该Canvas大小可变，强制使它充满网格的第一行。

添加到Canvas中的Image控件被固定在Canvas的左上角，这样可以确保窗口大小改变时，图像保持不变。随后将图片的高度设置为固定值，而宽度保持自由。由于将Stretch属性设置为UniformToFill，因此Image控件会将高度作为宽高比的标准，它可以自动调整自己的宽度来匹配已经设定好的高度和宽高比。

第一行的最后一个部分添加了一个Label控件，将其固定到Canvas的右上角，以确保调整窗口大小时，Label会随着右边缘移动。

接下来开始处理第二行，其中包含另一个Canvas控件，该控件中又包含一个Label控件。

底部的Canvas与此类似，所不同的是在其中添加的是一个Button控件，并将其固定到Canvas的右下角。这样可确保窗口大小改变时，该按钮始终位于窗口的右下角。OK文本前加下划线“_”即可为该按钮创建Alt+O快捷键。

最后在第三行中添加了一个StackPanel，再在其中添加Label和TextBlock控件。通过将第一个Label的Padding值设置为20,20,0,0，让该控件的内容距离上边缘和左边缘各20像素。

下一个Label的Padding值为20,0,0,0，只留出了左边的空白，这是因为两个Label之间的空间正好合适，并不需要多余的空白。

最后对TextBlock进行了设置。将属性TextWrapping设置为Wrap，以便文本在一行中容纳不下时自动换行。在窗口大小变化，一行变得更长后，文本又可以自动排列为较少行。这里也用到Margin和Padding属性。Margin设置为距离上方的Label控件10像素，Padding则将其内容设置为距离控件左边20像素。

最后，事件处理程序中的代码关闭窗口。在本例中，这相当于关闭整个应用程序，因为在第（19）步中将启动窗口设置为About窗口，因此关闭它就会自动关闭应用程序。

14.4.2 Options窗口

接下来将创建Options窗口。该窗口让玩家可设置一些可以改变游戏玩法的参数。其中也会用到一些之前未涉及的控件，包括CheckBox、RadioButton、ComboBox、TextBox和TabControl。

图14-10显示Options窗口中选中第一个选项卡时的情景。乍看起来，这个窗口和About窗口很像，但是我们在其中可以获得更多功能。

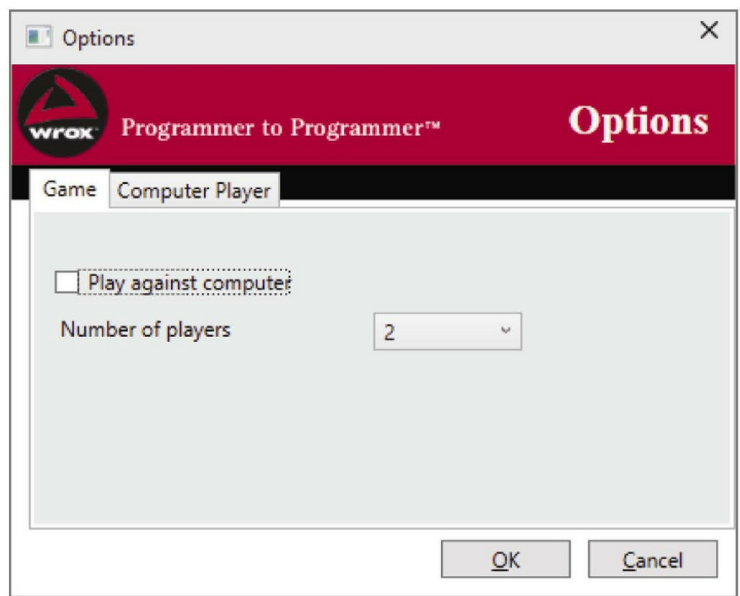


图14-10

1. TextBox控件

本章前面用过Label和TextBlock控件。这两个控件的作用只是向用户显示文本而已。而TextBox控件则允许用户向应用程序中输入一些文本。尽管这个控件也可以仅显示文本，但我们不应该单纯为了显示文本而使用它，除非在此基础上还允许用户编辑显示的文本。如果非要用TextBox来仅显示文本，需要将IsEnabled属性设置为false，以防用户编辑其中的内容。

使用表14-3中所示的一系列属性，可以控制在TextBox中输入和显示文本的方式。

表14-3 TextBox控件的属性

属性	说明
----	----

Text	TextBox控件中当前显示的文本
IsEnabled	将该属性设置为true时，用户可以编辑TextBox中的文本。如果为false，文本会显示为灰色，用户无法将键盘焦点放到该控件上
TextWrapping	某些情况下，我们希望TextBox只显示一行文本。这种情况下，可以将该属性值设置为NoWrap。这是默认值。如果希望将文本显示为多行，可将其值设置为Wrap或WrapWithOverflow。Wrap表示超出文本框边缘的文本内容会被移到下一行中。WrapWithOverflow则表示如果文本中没有合适的换行位置，允许非常长的单个单词超出文本框的边缘
VerticalScrollBarVisibility	如果允许用户在TextBox中输入多行文本，那么用户输入的内容有可能会超出文本框的下边界，从而无法完整显示。这种情况下，有必要使用滚动条进行操作。如果希望仅当文本过长时自动显示滚动条，可将此属性设置为Auto。设置为Visible表示始终显示滚动条，设置为Hidden或Disable则表示无论什么情况下都不显示滚动条
AcceptsReturn	此属性用于控制在TextBox控件中输入文本的方式。如果将其设置为默认值false，用户就不能通过回车键换行

2. CheckBox控件

CheckBox控件用于向用户显示可以选中或清除的选项。如果希望

向用户显示一个开关选项，或希望用户回答一个关于是否的问题，可以使用CheckBox控件。例如，在Options对话框中，我们希望用户选择是否要与电脑进行对战游戏。为此使用CheckBox控件，并在旁边标明文字“Play Against Computer”。

按照设计，CheckBox是独立实体，不会受到视图中其他CheckBox控件的影响。有时，我们会发现多个CheckBox有某种链接关系，选中其中一个后，其余的会被设置为未选中状态，但实际上这并不是CheckBox控件应有的用途。要实现这种功能，应该使用下一节介绍的RadioButton控件。

CheckBox也可以显示第三种状态，即“不确定”状态，表示不能回答“是”或“否”这个问题。当CheckBox用于显示其他项的信息时，经常使用这种状态。例如，CheckBox有时用于表示在一个树型视图中，是否所有子节点都已经被选中。这种情况下，如果所有节点都被选中，则CheckBox是选中状态；如果所有节点都未选中，则CheckBox为未选中状态；如果只选中了其中一部分节点，则CheckBox会是不确定状态。

表14-4列出了CheckBox控件常用的属性。

表14-4 CheckBox控件的属性

属性	说明
Content	CheckBox是一种内容控件，其中显示的内容是可以完全自定义的。在Content属性中添加一些文本会显示默认视图
IsThreeState	此属性用于指定该控件有两种状态还是三种状态。默认值为false，表示该控件只有两种状态

IsChecked	此属性的值可以是true或false。默认情况下，将其设置为true会显示为选中状态。如果IsThreeState为true，该属性还可以取值为null，表示该控件的状态为不确定
-----------	---

3. RadioButton控件

RadioButton总是与其他RadioButton控件结合使用，让用户可在多个选项中进行选择，并且某一时间只能选择一个选项。如果希望用户回答一些只有少数几种可选答案的问题，就可以使用RadioButton控件。而如果可能的答案多于4个，就需要考虑改用ListBox或ComboBox控件。在稍后创建的Options窗口中，用户可以选择电脑玩家的技能水平。我们设计了三种选项：**Dumb**（简单），**Good**（中等）和**Cheats**（很难）。当然，同一时刻只能选择一项。

如果在同一视图中要用到多个RadioButton控件，它们之间会默认建立一种关联，在其中一个被选中时，所有其余RadioButton控件都变为未选中状态。如果一个视图中的多个RadioButton控件不需要建立起这种关联，可将它们分到不同的组中，以免其他控件将这些没有关联的控件的值清除。

可使用表14-5中所示的属性来控制RadioButton。

表14-5 RadioButton控件的属性

属性	说明
Content	RadioButton是内容控件，因此可以修改其显示的内容。默认情况下，在Content中输入文本

IsChecked	值可以是true或false。如果IsThreeState被设置为true，还可以取值为null，表示状态不确定
GroupName	表示相应控件属于哪一组。默认情况下该属性的值为空，而GroupName值为空的所有RadioButton控件都被认为属于同一组

4. ComboBox控件

与RadioButton和CheckBox控件一样，ComboBox允许用户选择一个选项。不过，ComboBox与其存在两方面的根本性区别：

- ComboBox在一个下拉列表中显示可选项。
- ComboBox允许用户自行输入新值。

ComboBox常用于显示一个包含许多值的列表，例如国家或省的列表，但它们也可用于其他许多用途。在Options对话框中，ComboBox用于让用户选择玩家数量。尽管通过RadioButton也可以完成这个功能，但使用ComboBox可以节省视图空间。

ComboBox可以改为在其顶部显示一个Textbox，以便允许用户输入一些未能包含在列表中的值。本章中的一个练习要求在Options对话框中添加一个ComboBox控件，让用户既可以输入自己的名字，又可以从列表选择一个现有的名字。

该控件的IsReadOnly和IsEditable属性对于控件行为非常重要，将这两个属性结合起来使用，可以让用户通过4种不同方式使用键盘来选择ComboBox的值（见表14-6）。

表14-6 IsReadOnly和IsEditable属性的组合

	IsReadOnly 为 true	IsReadOnly 为 false
IsEditable 为 true	TextBox 正常显示，但控件本身对按键操作不会有任何反应。如果在列表中选择某一项，可在 TextBox 中选择文本	TextBox 正常显示，用户也可以正常进行输入。如果用户输入的内容已经在列表中，就会选中这部分内容。在用户输入内容的过程中，控件将显示该内容在列表中的最佳匹配项
IsEditable 为 false	如果 IsEditable 的值为 false，那么 IsReadOnly 的值不会有任何影响，因为不会显示文本框。选中该控件后，用户可通过输入方式选择列表中的某一项，却不能输入列表中不存在的值	

ComboBox是项控件，也就是说，我们可在其中添加许多项内容。表14-7中列举了ComboBox控件中的其他一些属性。

表14-7 ComboBox控件的其他属性

属性	说明
Text	Text属性表示要在ComboBox顶端显示的文本内容。可以是列表中的某一项，也可以是用户输入的新文本
SelectedIndex	表示选中的项在列表中的索引值。如果等于-1，代表没有进行任何选择，或者用户输入的内容不是列表中的某一项
SelectedItem	表示列表中实际的某一项，而不仅是索引值或文本内容。如果没有选择任何一项或者用户输入了新内容，返回null

5. TabControl控件

TabControl与本节中介绍的所有控件存在本质性差异。它是一个布

局控件，用于对页面上可以通过单击来选择的内容进行分组。

当希望在一个窗口中显示许多内容，又不希望让视图变得太杂乱时，可以使用TabControl。这种情况下，应该将不同的信息按照相关性分为不同的组，并为每一组建一个页面。一般来说，不应当让某一页中的控件影响到其他页面中的控件。如果它们互相影响，用户将不知道另一页中的选项已经被改变，导致他们产生困惑。

默认情况下，每个页面都由TabItem组成，TabItem又默认包含一个Grid控件。不过，可以根据需要把Grid替换为任何其他控件。在每个选项卡中都可以放置一些UI元素，并通过选择TabItem来切换不同的选项卡。每个TabItem都有一个用于显示选项卡名称的标题（Header）。该标题可以是一个Content控件，也就是说，可以自定义标题的显示内容，而不只是使用单纯的文字。

试一试：设计Options窗口：KarliCards Gui\Options.xaml

第一眼看到Options窗口时，也许会发现，它与About窗口十分相似，事实的确如此。正由于它们很像，所以我们可以重复使用之前示例中用到的一些代码。

- （1）在Solution Explorer中右击项目，选择Add|Window。将窗口命名为Options.xaml。
- （2）删除默认插入的Grid控件。
- （3）打开之前描述过的About.xaml窗口，将Grid控件及其中所有内

容复制并粘贴到新Options. xaml文件中。

(4) 修改窗口属性，如下所示：

```
Title="Options" Height="345" Width="434" ResizeMode="NoResize"
```

(5) 删除StackPanel及其所有内容。

(6) 删除Grid.Row属性值为3的Canvas控件及其所有内容。

(7) 删除Grid.Row属性值为1的Canvas控件中的Label控件。

(8) 修改Grid.Row属性值为0的Canvas控件中的Label控件，如下所示：

```
<Label Canvas.Right="10" Canvas.Top="13" Content="Options" FontFamily="Times New Roman" FontSize="24" FontWeight="Bold" /
```

(9) 将一个StackPanel控件拖曳到最底部的行，将其属性设置为：

```
Grid.Row="3" Orientation="Horizontal" FlowDirection="RightToLeft"
```

(10) 在StackPanel中添加两个按钮，如下所示：

```
<Button Content="_Cancel" Height="22" Width="75" Margin="10,0,0,10" Name="cancelButton" />
<Button Content="_OK" Height="22" Width="75" Margin="10,0,0,10" Name="okButton" />
```

(11) 将一个TabControl控件拖曳到第二行，将其属性设置为：

```
Grid.RowSpan="2" Canvas.Left="10" Canvas.Top="2" Width="408" +  
Grid.Row="1"
```

(12) 将两个TabItem控件的Header属性分别改为Game和Computer Player。

现在，该窗口的外观应该如图14-11所示，接下来就该在选项卡中插入一些内容了。

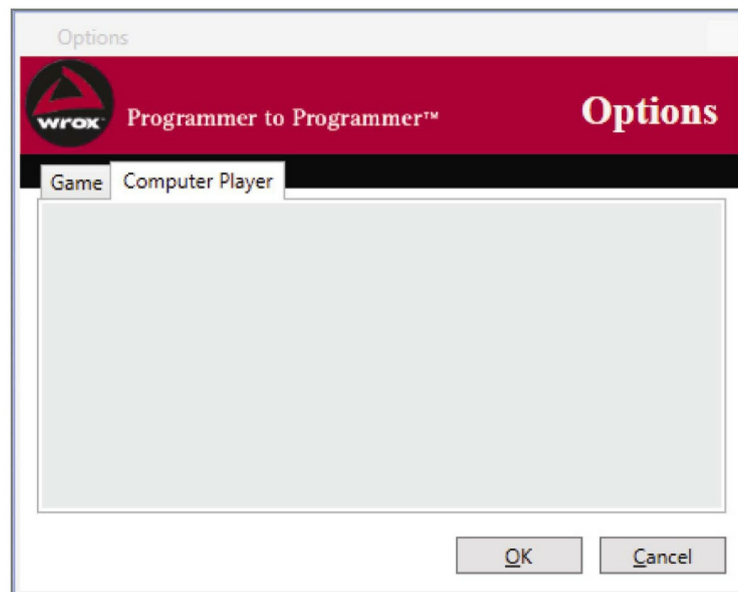


图14-11

(13) 选择Game TabItem，然后将一个CheckBox拖曳到其中。将其属性设置为：

```
Content="Play against computer" HorizontalAlignment="Left" Mar
```

```
VerticalAlignment="Top" Name="playAgainstComputerCheck"
```

(14) 在该TabItem中拖入一个Label控件和一个ComboBox控件，并将它们的属性设置为：

```
<Label Content="Number of players" HorizontalAlignment=
Margin="10,54,0,0" VerticalAlignment="Top" />
<ComboBox HorizontalAlignment="Left" Margin="196,58,0,0
VerticalAlignment="Top" Width="86" Name="numberOfPlayersComboE
SelectedIndex="0" >
    <ComboBoxItem>2</ComboBoxItem>
    <ComboBoxItem>3</ComboBoxItem>
    <ComboBoxItem>4</ComboBoxItem>
</ComboBox>
```

(15) 选择标题为Computer Player的第二个TabItem。将一个Label和三个RadioButton控件拖曳到Grid中，然后将它们的属性设置为：

```
<Label Content="Skill Level" HorizontalAlignment="Left"
Margin="10,10,0,0" VerticalAlignment="Top"/>
<RadioButton Content="Dumb" HorizontalAlignment="Left"
Margin="37,41,0,0" VerticalAlignment="Top" IsChecked="True"
Name="dumbAIRadioButton"/>
<RadioButton Content="Good" HorizontalAlignment="Left"
Margin="37,62,0,0" VerticalAlignment="Top" Name="goodAIRadioBu
    <RadioButton Content="Cheats" HorizontalAlignment="Left"
Margin="37,83,0,0" VerticalAlignment="Top"
Name="cheatingAIRadioButton"/>
```

(16) 至此已经完成该窗口的布局。打开App.xaml文件，将StartupUri设置为Options.xaml。

(17) 运行该应用程序。

示例的说明

窗口的ResizeMode设置为NoResize。这样，我们在放置控件时就不必考虑窗口大小改变时会发生什么情况了，因为用户无法调整窗口的大小。

第(9)步中的StackPanel引入了新属性FlowDirection，其值为RightToLeft。这样，添加到其中的两个按钮就会靠紧对话框的右边缘，而不是默认的左边缘了。有趣的是，这样的修改也会改变两个按钮的Margin属性的含义，即Left和Right的含义会互换。

第二个选项卡中的RadioButtons并未指定GroupName，这样，它们就会作为一组。随后为其中第一个设置了IsChecked属性为true，使其成为默认的选中项。

6. 处理Options窗口中的事件

现在，Options窗口看起来已经不错了，但用户还不能通过它实现什么功能，即使更改其中的设置也没有任何意义。用户希望他们所做的选择可以保存下来，并且被应用程序使用。为此，可将控件的值保存在这个窗口中，但是这样做之后会非常缺乏灵活性，会将应用程序数据与

GUI混在一起，因此并不是一种好的设计。我们应该创建一个类，并将用户所做的选择保存在其中。

试一试：处理事件：KarliCards Gui\Options.xaml

本例会在项目中添加一个新类，将用户所做的选择保存在其中，并处理用户改变选择时所发生的事件。

（1）在项目中新建一个类，将其命名为GameOptions.cs。

（2）输入如下代码：

```
using System;
namespace KarliCards_Gui
{
    [Serializable]
    public class GameOptions
    {
        public bool PlayAgainstComputer { get; set; }
        public int NumberOfPlayers { get; set; }
        public int MinutesBeforeLoss { get; set; }
        public ComputerSkillLevel ComputerSkill { get; set; }
    }
    [Serializable]
    public enum ComputerSkillLevel
    {
```

```

        Dumb,
        Good,
        Cheats
    }
}

```

(3) 返回代码隐藏文件Options.xaml.cs，添加一个private字段来保存GameOptions实例：

```
private GameOptions _gameOptions;
```

(4) 在构造函数中添加以下代码：

```

using System.IO;
using System.Windows;
using System.Xml.Serialization;
namespace KarliCards_Gui
{
    ///<summary>
    /// Interaction logic for Options.xaml
    ///</summary>
    public partial class Options : Window
    {
        private GameOptions _gameOptions;
        public Options()
        {
            if (_gameOptions == null)

```



```
{
```

```
    if (File.Exists("GameOptions.xml"))
```

```
    {
```

```
        using (var stream = File.OpenRead("GameOptions.xml"))
```

```
        {
```

```
            var serializer = new XmlSerializer(typeof(GameOpti
```

```
            _gameOptions = serializer.Deserialize(stream) as G
```

```
}
```

```
}
```

```
else
```

```
    _gameOptions = new GameOptions();
```

```
}
```

```
InitializeComponent();
```

```
}
```

(5) 转到设计视图，分别双击三个RadioButton控件，在代码隐藏文件中添加Checked事件处理程序。按照如下所示修改处理程序：

```
private void dumbAIRadioButton_Checked(object sender, RoutedEventArgs)
{
    _gameOptions.ComputerSkill = ComputerSkillLevel.Dumb;
}
private void goodAIRadioButton_Checked(object sender, RoutedEventArgs)
{
    _gameOptions.ComputerSkill = ComputerSkillLevel.Good;
}
private void cheatingAIRadioButton_Checked(object sender, RoutedEventArgs)
{
    _gameOptions.ComputerSkill = ComputerSkillLevel.Cheats;
}
```

(6) 返回设计视图，在Game选项卡中选择TextBox。在Properties面板中单击闪电图标，随后双击GotFocus事件，以便在代码隐藏文件中添加处理程序。

(7) 输入以下代码：

```
private void timeAllowedTextBox_GotFocus(object sender, RoutedEventArgs)
{
    timeAllowedTextBox.SelectAll();
}
```

(8) 再次在设计视图中选择TextBox，然后在代码隐藏文件中添加

PreviewMouseLeftButtonDown事件处理程序。

(9) 输入如下代码：

```
private void timeAllowedTextBox_PreviewMouseLeftButtonDown(obj
    MouseButtonEventArgs e)
{
    var control = sender as TextBox;
    if (control == null)
        return;
    Keyboard.Focus(control);
    e.Handled = true;
}
```

(10) 运行该应用程序。

示例的说明

新类目前仅是一系列可保存Options窗口中各种值的属性。我们将其标记为Serializable，以便保存为一个文件。

当用户选中RadioButton时，会发生Checked事件。我们对该事件进行处理，以便设置GameOptions实例的ComputerSkillLevel属性值。

14.4.3 数据绑定

数据绑定是一种以声明方式将控件与数据关联到一起的方法。在Options窗口中，通过处理RadioButton的Checked事件，来设置GameOptions类的ComputerSkillLevel属性值。这种处理方式没什么问题，我们可以通过代码和事件处理程序来设置窗口中的所有值，但通常，更好的办法是直接将控件的属性与对应的数据绑定起来。

一个绑定（Binding）关系由4个组件构成：

- 绑定目标：指定绑定要应用到的对象
- 目标属性：指定要设置的属性
- 绑定源：指定绑定使用的对象
- 源属性：指定存储该数据的属性

不需要在每次使用时都明确指定这4个组件。特别是，由于设置的是绑定到控件的一个属性，因此通常绑定目标已经被隐式指定。

总是要设置绑定源，而后才能使绑定关系正常运作起来，只不过其设置方式多种多样。在接下来的小节和第15章中，将介绍绑定源数据的几种不同方法。

1. DataContext控件

DataContext控件用于定义一个数据源，该数据源可以绑定某个元素的所有子元素。很多时候，经常用类的一个实例来保存视图中的大部分数据。这种情况下，可将窗口的DataContext设置为该对象的实例，从而可以将该类与视图中的属性绑定起来。该方法将在“动态绑定到外部对象”小节中介绍。

2. 绑定到本地对象

可绑定到任何包含所需数据的.NET对象，只要编译器能够定位该对象即可。如果在使用对象的控件所在的上下文环境（即相同的XAML代码块）中可以找到该对象，就可通过设置绑定的ElementName属性来指定绑定源。请看对Options窗口中的ComboBox控件所做的更改：

```
<ComboBox HorizontalAlignment="Left" Margin="196,58,0,0" VerticalAlignment="Top" Width="86" Name="numberOfPlayersComboBox" SelectedIndex="0" IsEnabled="{Binding ElementName=playAgainstComputerCheck, Path=IsChecked}" />
```

注意IsEnabled属性。没有指定true或false值，而是使用了一长串用花括号括起来的文本。这种指定属性值的方法称为“标记扩展语法”，也是一种用于指定属性的便捷方法。还可以使用以下写法：

```
<ComboBox HorizontalAlignment="Left" Margin="196,58,0,0, VerticalAlignment="Top" Width="86" Name="numberOfPlayersComboBox" SelectedIndex="0" >
    <ComboBox.IsEnabled>
        <Binding ElementName="playAgainstComputerCheck" Path="IsChecked" />
    </ComboBox.IsEnabled>
</ComboBox>
```

上面两段示例代码都可将绑定源设置为playAgainstComputerCheck复选框。源属性是通过Path指定的IsChecked属性。

绑定目标被设置为IsEnabled属性。两段示例代码都通过将绑定指定为该属性的内容来完成这种设置，只不过使用了两种不同的语法而已。

最后，由于在ComboBox上进行绑定，因此也就隐式指定了绑定目标。

本例中的这一绑定关系可以让ComboBox的IsEnabled属性随着CheckBox的IsChecked属性值自动进行设置或清除。结果，我们没有使用任何代码，就可以在用户更改CheckBox的值时启用和禁用ComboBox。

3. 静态绑定到外部对象

通过在XAML中将某个类指定为一项资源，就可以动态创建对象实例。具体的方法就是首先在XAML中添加相应的名称空间，以便可以找到这个类，然后在XAML的某个元素中将类声明为资源。

可在希望进行数据绑定的对象的父元素中创建资源引用。

**试一试：创建静态数据绑定：KarliCards
Gui\NumberOfPlayers.cs**

在本例中，将新建一个用来保存Options窗口中ComboBox数据的新类，并将其与该控件绑定起来。

(1) 在项目中新建一个类，并将其命名为NumberOfPlayers.cs。

(2) 添加如下代码：

```
using System.Collections.ObjectModel;
```

```

namespace KarliCards_Gui
{
    public class NumberOfPlayers : ObservableCollection<int>
    {
        public NumberOfPlayers()
        : base()
        {
            Add(2);
            Add(3);
            Add(4);
        }
    }
}

```

(3) 返回Options.xaml文件的设计视图，选择Window根元素。

(4) 选择包含ComboBox的Canvas元素，并将下列代码添加到其下方，但要在TabControl声明之前：

```

<Canvas.Resources>
    <local:NumberOfPlayers x:Key="numberOfPlayersData" />
</Canvas.Resources>

```

(5) 选择ComboBox，并从中删除三个ComboBoxItem。

(6) 在其中添加属性：

```

ItemsSource="{Binding Source={StaticResource numberOfPlayersDa

```


示例的说明

在本例中，我们完成了多项工作。`NumberOfPlayers`类继承自一个特殊集合`ObservableCollection`。这个基类是一个进行过扩展的集合，以使其能在WPF中更好地发挥作用。在该类的构造函数中，我们为该集合添加了几个值。

接下来在Canvas中新建了一个资源。其实可在ComboBox的任意父元素中创建这个资源。一旦在元素中指定了某个资源，它的所有子元素就都可以使用这一资源。

最后通过`ItemsSource`设置了绑定关系。`ItemsSource`属性被设计用于在项控件中，为项集合设置绑定。在绑定中，只需要指定绑定源。绑定目标、目标属性和源属性设置都是在`ItemsSource`属性中处理的。

4. 动态绑定到外部对象

现在，可绑定到根据需要动态创建的对象，以便为它们提供数据。在希望对一个现有实例化对象进行数据绑定时，我们应该使用什么方法呢？这种情况下，需要在代码中加一点料。

以Options窗口为例，我们并不希望其中的选项在每次打开窗口时都被清除，而是希望用户所做的选择可以被保存下来，并且可用在应用程序的其余部分。

在代码中将`DataContext`属性的值设置为此实例，可以实现上述功能。

试一试：创建动态绑定：**KarliCards Gui\GameOptions.cs**

在本例中，我们会将Options窗口中其余的控件与GameOptions实例绑定起来。

(1) 打开Options.xaml.cs代码隐藏文件。

(2) 在构造函数的底部，在InitializeComponent()这一行之前添加以下代码：

```
DataContext = _gameOptions;
```

(3) 转到GameOptions类，对其进行修改，如下所示：

```
using System;
using System.ComponentModel;
namespace KarliCards_Gui
{
    [Serializable]
    public class GameOptions : INotifyPropertyChanged
    {
        private bool _playAgainstComputer = true;
        private int _numberOfPlayers = 2;
        private ComputerSkillLevel _computerSkill = ComputerSkillLevel.Beginner;
        public int NumberOfPlayers
        {
```

```

    get { return _numberOfPlayers; }
    set
    {
        _numberOfPlayers = value;
        OnPropertyChanged(nameof(NumberOfPlayers));
    }
}

public bool PlayAgainstComputer
{
    get { return _playAgainstComputer; }
    set
    {
        _playAgainstComputer = value;
        OnPropertyChanged(nameof(PlayAgainstComputer));
    }
}

public ComputerSkillLevel ComputerSkill
{
    get { return _computerSkill; }
    set
    {
        _computerSkill = value;
        OnPropertyChanged(nameof(ComputerSkill));
    }
}

public event PropertyChangedEventHandler PropertyChanged
private void OnPropertyChanged(string propertyName)

```

```

        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs
        }
    }
}
[Serializable]
public enum ComputerSkillLevel
{
    Dumb,
    Good,
    Cheats
}
}

```

(4) 返回Options.xaml文件，选择CheckBox，然后添加IsChecked属性，如下所示：

```
IsChecked="{Binding Path=PlayAgainstComputer}"
```

(5) 选择ComboBox，然后按照如下方式进行修改，删除SelectedIndex属性，修改ItemsSource和SelectedValue属性：

```

<ComboBox HorizontalAlignment="Left" Margin="196,58,0,0" Verti
Width="86" Name="numberOfPlayersComboBox"
ItemsSource="{Binding Source={StaticResource numberOfPlayersDa
SelectedValue="{Binding Path=NumberOfPlayers}" />

```

(6) 选中并双击OK按钮，在代码隐藏文件中为其添加Click事件处理程序。使用相同的步骤为Cancel按钮添加相应的处理程序：

```

private void okButton_Click(object sender, RoutedEventArgs
{
    using (var stream = File.Open("GameOptions.xml", FileMode
    {
        var serializer = new XmlSerializer(typeof(GameOptions));
        serializer.Serialize(stream, _gameOptions);
    }
    Close();
}
private void cancelButton_Click(object sender, RoutedEventArgs
{
    _gameOptions = null;
    Close();
}

```

(7) 运行该应用程序。

示例的说明

将窗口的DataContext设置为GameOptions实例后，可以通过指定绑定中使用的属性很方便地绑定到该实例。这就是在第（4）、（5）步中所做的。需要注意，ComboBox是通过一个静态资源中的项来填充的，但选定的值在GameOptions实例中设置。

GameOptions发生了较大变化。它实现了INotifyPropertyChanged接口，也就是说，当属性值发生变化时，这个类就会通知WPF。为让这个通知生效，我们需要让订阅方调用上述接口中定义的PropertyChanged事

件。为此，属性设置器必须主动对它们进行调用，这一调用是通过辅助方法OnPropertyChanged来实现的。

调用OnPropertyChanged方法时，使用了C# 6引入的一个新表达式:nameof。通过一个表达式调用nameof(...)时，它将检索最终标识符的名称。这在OnPropertyChanged方法中特别有用，因为它把要更改的属性名作为一个字符串。

OK按钮的事件处理程序使用XmlSerializer将设置保存到磁盘中，而Cancel事件处理程序将GameOptions字段设置为null，这样可以确保用户所做的选择可以被清除掉。这两个事件处理程序都会执行关闭窗口操作。

14.4.4 使用ListBox控件启动游戏

现在，在游戏中，我们只剩下一个提供支持的窗口需要创建了。在创建游戏主界面之前，最后一个窗口用于让玩家添加新的玩家，以及指定在新一轮游戏中有哪些玩家需要加入。该窗口使用一个ListBox控件来显示玩家的名字。

通常，ListBox和ComboBox控件的作用是类似的，只不过ComboBox控件一般只能选择一项，而ListBox允许用户选择多项。另一个显著差异是ListBox控件用于显示其内容的列表总处于展开状态。也就是说，ListBox控件会占用窗口中更多的空间，但用户可以立即看到相应的选项。

表14-8中列出了ListBox控件一些比较重要的属性。

表14-8 **ListBox**控件的重要属性

属性	说明
SelectionMode	该属性控制用户在列表中进行选择的方式。可以有三种取值： Single ，只允许用户选择一项； Multiple ，允许用户不必按下Ctrl键即可选择多项； Extended ，允许用户通过按下Shift键选择连续的多项，或者按下Ctrl键选择非连续的多项
SelectedItem	获取或设置第一个被选中的项，如果没有被选项，返回null。即使有多项被选中，也仅返回第一项
SelectedItems	获取包含当前所有已选中项的列表
SelectedIndex	与 SelectedItem 类似，不同之处在于仅返回所选项的索引值，而不是项本身。如果没有被选项，返回-1，而不是null

试一试：创建启动游戏窗口：**KarliCards** **Gui\StartGame.xaml**

该窗口会在新游戏开始之前显示给用户。用户可以在其中输入自己的名字，也可以从已知玩家的列表中选择已经存在的名字。

(1) 新建一个窗口，将其命名为**StartGame.xaml**。

(2) 删除该窗口中的Grid元素，并将**Options.xaml**窗口中的主Grid元素及其内容复制到新建的窗口中。

(3) 将Grid.Row属性值为1的那个Canvas控件中的所有内容删除。

(4) 将窗口标题修改为“Start New Game”，并设置以下属性：

```
Height="345" Width="445" ResizeMode="NoResize"
```

(5) 将网格第一行（编号为0）中Label的内容修改为“New Game”。

(6) 打开GameOptions.cs文件，并将下列字段添加到该类的顶部：

```
private ObservableCollection<string> _playerNames =  
new ObservableCollection<string>();  
public List<string> SelectedPlayers { get; set; }
```

(7) 上面这段代码用到了System.Collections.Generic和System.Collections.ObjectModel名称空间，所以使用以下语句：

```
using System.Collections.Generic;  
using System.Collections.ObjectModel;
```

(8) 添加一个构造函数，以便初始化SelectedPlayers集合：

```
public GameOptions()  
{  
    SelectedPlayers = new List<string>();  
}
```

(9) 为该类添加一个属性和两个方法，如下所示：

```
public ObservableCollection<string> PlayerNames
```



```

{
    get
    {
        return _playerNames;
    }
    set
    {
        _playerNames = value;
        OnPropertyChanged("PlayerNames");
    }
}

public void AddPlayer(string playerName)
{
    if (_playerNames.Contains(playerName))
        return;
    _playerNames.Add(playerName);
    OnPropertyChanged("PlayerNames");
}

```

(10) 返回StartGame.xaml窗口。

(11) 在Canvas的下方，即网格行1中添加一个ListBox控件、两个Label控件、一个TextBox控件以及一个Button控件，并按照图14-12所示的布局 and 外观修改这些控件。

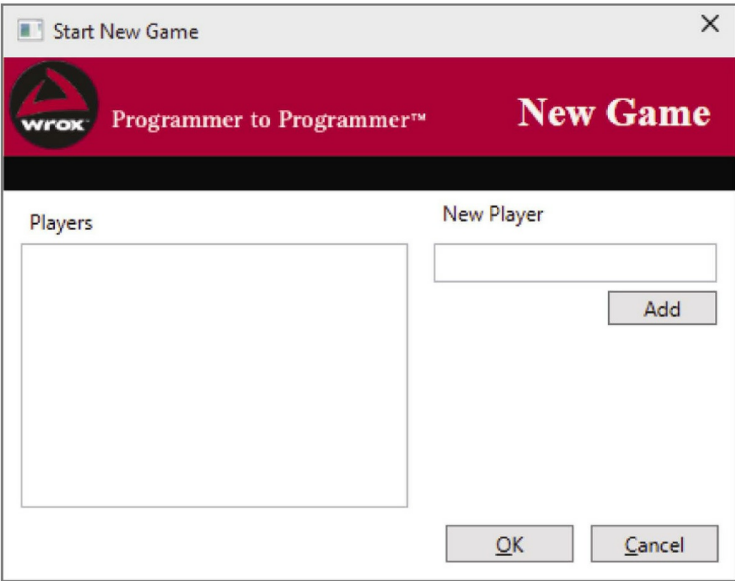


图14-12

（12）按照表14-9所示设置控件的Name属性。

表14-9 Name属性

控件	Name
TextBox	newPlayerTextBox
Button	addNewPlayerButton
ListBox	playerNamesListBox

（13）设置ListBox的ItemSource属性，如下所示：

```
ItemSource="{Binding Path=PlayerNames}"
```

（14）在代码隐藏文件中为ListBox添加SelectionChanged事件处理程序，并添加如下代码：

```
private void playerNamesListBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{

    if (!_gameOptions.PlayAgainstComputer)

        okButton.IsEnabled = (playerNamesListBox.SelectedItems.
Count > 0);

    else

        okButton.IsEnabled = (playerNamesListBox.SelectedItems.
Count > _gameOptions.NumberOfPlayers);

}
```

(15) 将OK按钮的IsEnabled属性设置为false。

(16) 打开代码隐藏文件，并在该类的开头添加以下字段：

```
private GameOptions _gameOptions;
```

(17) 从Options.xaml.cs代码隐藏文件中复制构造函数（注意不要仅复制其名称），并在代码末尾的InitializeComponent之后添加下列内容（别忘了为System.IO和System.Xml.Serialization添加using声明）：

```
if (_gameOptions.PlayAgainstComputer)
    playerNamesListBox.SelectionMode = SelectionMode.Single;
else
    playerNamesListBox.SelectionMode = SelectionMode.Extended;
```

(18) 选择Add按钮，并为其添加Click事件处理程序。添加如下代码：

```
private void addNewPlayerButton_Click(object sender, RoutedEventArgs e)
{
    if (!string.IsNullOrEmpty(newPlayerTextBox.Text))
    {
        _gameOptions.AddPlayer(newPlayerTextBox.Text);
    }
}
```

```
newPlayerTextBox.Text = string.Empty;
```

```
}
```

(19) 将Options.xaml.cs代码隐藏文件中OK和Cancel按钮的事件处理程序代码复制到当前这个代码隐藏文件中。

(20) 在OK按钮的事件处理程序的开头添加以下几行代码：

```
foreach (string item in playerNamesListBox.SelectedItems)
{
    _gameOptions.SelectedPlayers.Add(item);
}
```

(21) 转到App.xaml文件，将StartupUri设置为StartGame.xaml。

(22) 运行该应用程序。

示例的说明

首先在GameOptions类中添加了一些代码，使其可以保存所有已知玩家的信息以及用户在StartGame窗口中所做的当前选择。

ListBox的ItemsSource属性与之前在ComboBox中看到的类似。不

过，与之前将ComboBox中的所选项直接与某个值进行绑定不同的是，对ListBox进行绑定要复杂一些。如果尝试绑定SelectedValues属性，我们会发现，这个属性其实是只读的，不能直接用于数据绑定。这里采用的解决办法是通过编写代码，使用OK按钮来保存相应的值。需要注意，这里可以强制转换为IList<string>，是因为此时ListBox的内容是字符串，但如果我们选择修改默认行为，显示其他内容，那么所选择的项也必须进行更改。

当ListBox中的已选项发生变化时，就会触发SelectionChanged事件。此时，我们要处理该事件，以便确定所选项的数目是否正确。如果玩家选择与电脑进行对战，那么只有一个真正的玩家；否则，必须选择相应数量的玩家名字。

注意：第15章将介绍样式、控件和项模板，还将介绍为什么我们不能随时了解控件的内容类型是什么。

14.5 练习

(1) **TextBlock**控件可用来显示大量文本内容，但如果文本内容超过了控件区域的大小，那么并不提供滚动功能。请将**TextBlock**和另一种控件结合起来，创建一个可以包含大量文本内容，并且仅当文本内容超出显示区域时才会出现滚动条的窗口。

(2) **Slider**和**ProgressBar**控件具有一些相同的属性，例如最小值、最大值和当前值。仅在**ProgressBar**控件上使用数据绑定，创建一个包含**Slider**和**ProgressBar**的窗口，且**Slider**控件可以控制**ProgressBar**的最小值、最大值和当前值。

(3) 将前一题中的**ProgressBar**控件修改为从窗口左下角到右上角沿对角线显示。

(4) 新建一个名为**PersistentSlider**，且包含**MinValue**，**MaxValue**和**CurrentValue**三个属性的类。这个类应该支持数据绑定，并且所有属性都可以将更改通知给绑定的控件。

a. 在前两个练习所创建的窗口的代码隐藏文件中新建一个**PersistentSlider**类型的字段，并使用一些默认值对其进行初始化。

b. 在构造函数中将该实例绑定到窗口数据源。

c. 将**Slider**的**Minimum**、**Maximum**和**Value**属性绑定到数据源。

附录A给出了练习答案。

14.6 本章要点

第15章 高级桌面编程

本章内容：

- 如何使用路由命令来代替事件
- 如何使用Menu控件和路由命令来创建菜单
- 如何使用XAML样式来设置控件和应用程序的样式
- 如何创建值转换器
- 如何使用时间线来创建动画
- 如何定义和引用静态及动态资源
- 如何在常用控件不满足需要时创建用户控件

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 15 Code后，可找到与本章示例对应的单独文件。

到目前为止，我们使用WPF的方式与Visual Studio中创建Windows应用程序的另一种主流技术——Windows Forms十分类似。但接下来介绍的内容就有所不同了。WPF可设置所有控件的样式，使用模板让现有控件看起来不再是原生的外观。此外，我们还将直接输入XAML，学习

更多知识。尽管编写XAML一开始看起来很难，而通过设置属性来移动或调整控件的外观很容易上手，XAML可以实现许多在设计器中无法实现的功能，例如创建动画。

下面接着第14章的内容，继续之前的游戏客户端开发。

15.1 主窗口

该应用程序的主窗口是玩游戏时的主界面，而现在其中还没有太多控件。本章将开发这个游戏，但在开始之前，还必须做三件事。首先给项目添加主窗口，然后在其中添加菜单，最后将已构建好的窗口与菜单项绑定起来。

15.1.1 菜单控件

大多数应用程序都包含某类菜单和工具栏。它们的目的是相同的：让用户轻松地浏览应用程序的内容。工具栏通常包含菜单所提供的相同菜单项的子集，可将其视为菜单项的快捷方式。

Visual Studio内置了Menu和Toolbar控件。下面将介绍Menu控件的用法，Toolbar控件的用法与其非常类似。

默认情况下，菜单项显示为水平的一栏，每个菜单项都可以展开其下拉菜单。菜单是Item控件，所以，可以修改其包含的默认内容；不过，一般使用某种形式的MenuItem（菜单项），如下例所示。每个MenuItem都可以包含其他菜单项，将MenuItem嵌套起来，就可以建立复杂的菜单，但应使菜单结构尽可能简洁。

使用一些属性，可以控制MenuItem控件的显示方式（见表15-1）。

表15-1 MenuItem的显示属性

属性	说明
Icon	在控件的左侧显示一个小图标
IsCheckable	在控件的左侧显示一个CheckBox控件
IsChecked	获取或设置MenuItem上的Checkbox值

15.1.2 路由命令和菜单

路由命令（`routed command`）在第14章中简单介绍过，现在将第一次用到它。路由命令与事件类似，都是在用户执行某个操作时执行代码，都可以返回某个状态，表示它们在任何给定时间是否可以执行。

为什么使用路由命令而不使用事件，至少有三个理由：

- （1）在应用程序的多个不同位置触发某个事件的操作。
- （2）UI元素应只在特定条件下才可用，例如在没有内容需要保存时，保存按钮就应该禁用。
- （3）希望断开处理事件的代码和代码隐藏文件的联系。

如果出现上述几种情况，就可以考虑使用路由命令。对于本章开发的游戏，某些菜单项也应能通过工具栏来执行。还有，Save操作应只在游戏过程中可用，且应在菜单和工具栏中都可用。

注意： 重要的是，只有在KarliCards GUI项目中正确设置了默认

名称空间，才能使示例工作。如果出现了编译器错误，指出类或资源不是名称空间的成员，就可能使用了与本书不同的名称空间。

KarliCards解决方案使用了两个根名称空间：用于Ch13CardLib项目的Ch13CardLib和用于KarliCards GUI项目的KarliCards_Gui。如果出了问题，就尝试在整个项目中改变名称空间，来匹配本书使用的那些名称空间。

试一试：创建主窗口：**KarliCards Gui\MainWindow.xaml**

本例会为游戏创建主窗口。因为这是应用程序的主窗口，所以使用之前已创建好的窗口。

(1) 给项目添加一个新窗口，命名为GameClient.xaml。

(2) 将窗口的标题改为Karli Cards Game Client，并删除Height和Width属性。

(3) 将WindowState属性设置为Maximized。

(4) 添加以下名称空间：

```
xmlns:src="clr-namespace:KarliCards_Gui"
```

(5) 删除窗口中的网格，并将StartGame窗口中的网格及其所有内容复制过来。

(6) 在<Grid>标记中，将除了Grid.Row=0中的Canva和<Grid.RowDefinitions>之外的所有内容删除。

(7) 将一个DockPanel控件拖到网格中编号为1的行，设置其属性，如下所示：

```
Grid.Row="1" Margin="0"
```

(8) 选中DockPanel，并将一个Menu控件拖到它上面。这个Menu控件会占满DockPanel的整个区域，这正是我们需要的效果。

(9) 修改Menu控件的属性，使其背景为黑色，字体为粗体，前景色为白色，代码如下：

```
Background="Black" FontWeight="Bold" Foreground="White"
```

(10) 在设计视图中右击该Menu控件，并选择Add MenuItem命令。

(11) 将Header属性改为“_File”。注意单词前面要加一个下划线。再将其前景色设置为白色。

(12) 在_File项中再添加一个MenuItem，做法是右击_File项，然后选择“Add MenuItem”命令。设置其Height、Width、Header和Foreground属性：

```
<MenuItem Header="_File" Foreground="White">  
  <MenuItem Header="_New Game..." Height="22"  
    Width="200" Foreground="Black" />  
</MenuItem>
```

(13) 将下列MenuItem添加到File菜单中:

```
<MenuItem Header="_Open" Width="200" Foreground="Black"/>
<MenuItem Header="_Save" Width="200" Foreground="Black" Cor
  <MenuItem.Icon>
    <Image Source="Images\base_floppydisk_32.png" Width="20
  </MenuItem.Icon>
</MenuItem>
<Separator Width="145" Foreground="Black"/>
<MenuItem Header="_Close" Width="200" Foreground="Black" C
```

(14) 在File菜单项的同一级别添加以下MenuItem控件:

```
<MenuItem Header="_Game" Background="Black" Foreground="Wh:
  <MenuItem Header="_Undo" HorizontalAlignment="Left"
    Width="145" Foreground="Black"/>
</MenuItem>
<MenuItem Header="_Tools" Background="Black" Foreground="Wl
  <MenuItem Header="_Options" HorizontalAlignment="Left"
    Width="145" Foreground="Black"/>
</MenuItem>
<MenuItem Header="Help" Background="Black" Foreground="Whi
  <MenuItem Header="_About" HorizontalAlignment="Left"
    Width="145" Foreground="Black"/>
</MenuItem>
```

(15) 将主Grid控件的背景色设置为绿色。可以把背景颜色设置为标准的颜色,做法是单击Background框右边的颜色框,选择Custom

Expression。然后输入希望的颜色名字，本例是“Green”。

(16) 在第一个Grid控件之前，将以下命令绑定添加到窗口中：

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Close"
    CanExecute="CommandCanExecute" Executed="CommandExecuted"
  <CommandBinding Command="ApplicationCommands.Save"
    CanExecute="CommandCanExecute" Executed="CommandExecuted"
</Window.CommandBindings>
```

(17) 在Grid控件中，将编号为0的一行标签内容从“New Game”改为“Karli Cards”。

(18) 在编号为2的一行中添加一个新的Grid控件，命名为ContentGrid:

```
<Grid Grid.Row="2" x:Name="contentGrid" />
```

现在窗口应该如图15-1所示。

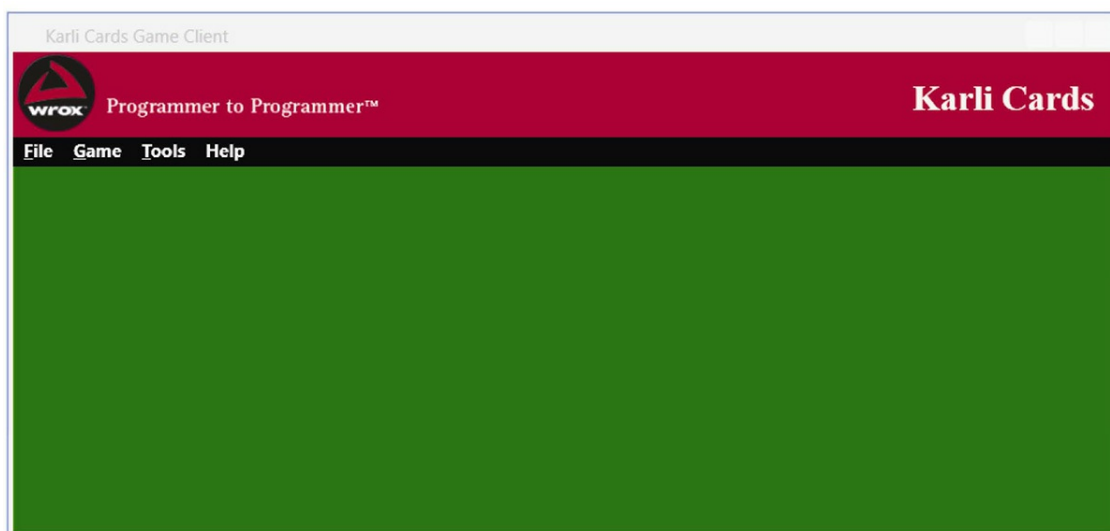


图15-1

(19) 打开GameClient.xaml.cs代码隐藏文件，添加下面两个方法：

```
private void CommandCanExecute(object sender, CanExecuteRoute
{
    if (e.Command == ApplicationCommands.Close)
        e.CanExecute = true;
    if (e.Command == ApplicationCommands.Save)
        e.CanExecute = false;
    e.Handled = true;
}
private void CommandExecuted(object sender, ExecutedRoute
{
    if (e.Command == ApplicationCommands.Close)
        this.Close();
    e.Handled = true;
}
```

(20) 在App.xaml文件中将StartupUri属性改为GameClient.xaml，然后运行该应用程序。

示例的说明

运行这个应用程序时，Game Client窗口一开始会最大化显示，且仍可以根据需要调整其大小。按住Alt键时，File菜单会获得焦点，F字母也会加上下划线，说明可以按F键展开该菜单。

展开菜单后，Save菜单处于禁用状态，但该菜单会显示一个磁盘图标，在元素标题的右边会显示“Ctrl+S”标注。这表示可按Ctrl+S快捷键来访问该菜单（当其可用时）。前面没有设置任何快捷键，为什么会有这样一个标注？实际上，为该菜单项设置命令的代码如下：

```
<MenuItem Header="_Save" Width="200" Foreground="Black" Commar
```

这个Save命令由WPF定义。File菜单中的Save和Close菜单项是在ApplicationCommands类中定义的，它还定义了Cut、Copy、Paste和Print菜单项。为MenuItem指定Save命令时，快捷键Ctrl+S就会分配给这个菜单项，因为大多数Windows应用程序都使用这个标准组合键来访问这个功能。

在代码隐藏文件中添加了两个方法，用于确定命令的状态及执行的操作。在XAML中，创建了两个命令绑定，来使用此类方法：

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Close"
        CanExecute="CommandCanExecute" Executed="CommandExecuted"
    <CommandBinding Command="ApplicationCommands.Save"
        CanExecute="CommandCanExecute" Executed="CommandExecuted"
</Window.CommandBindings>

private void CommandCanExecute(object sender, CanExecuteRou
{
    if (e.Command == ApplicationCommands.Close)
        e.CanExecute = true;
    if (e.Command == ApplicationCommands.Save)
        e.CanExecute = false;
```

```
        e.Handled = true;
    }
    private void CommandExecuted(object sender, ExecutedRoutedEvent
    {
        if (e.Command == ApplicationCommands.Close)
        {
            this.Close();
            e.Handled = true;
        }
    }
```

命令绑定中的CanExecute部分指定，调用一个方法来确定命令在当时是否对用户可用，Executed部分指定，方法应在用户激活命令时调用。注意，这与命令在何处激活无关。如果菜单项和按钮都包含了Save命令，那么绑定对它们都有效。

CommandCanExecute目前的实现代码太过简单，实际上应该进行一些计算，以确定应用程序是否准备好保存数据。因为游戏还没有需要保存的内容，所以只为Save命令返回false值。为此，设置CanExecuteRoutedEventArgs类的e.CanExecute属性。另一方面，Close命令可以正常执行，所以给它返回true。

CommandExecuted执行的测试与CommandCanExecute相同。如果确定要执行的命令是Close，就关闭当前窗口。

15.2 创建控件并设置样式

现在抛开游戏的客户端实现，更多地关注游戏本身。图形化纸牌游戏的一个关键元素是纸牌。显然，在WPF自带的标准控件中没有“纸牌”控件，所以只能自己创建它。

WPF的一个最佳特性是允许设计人员完全控制用户界面的外观和操作系统。其核心是可以根据需要设置控件的二维或三维样式。前面只使用.NET为控件提供的基本样式，但实际上可设置任意多种不同的样式。

本节介绍两个基本技术：

- 样式——批量设置要应用到控件上的某些属性
- 模板——在其基础上设置控件外观的控件

这两种技术有一些重叠，因为样式可以包含模板。

15.2.1 样式

WPF控件有一个Style属性（继承自FrameworkElement），它可以设置为Style类的实例。Style类相当复杂，可用来实现高级的样式功能，但其核心实际上也就是一组Setter对象。每个Setter对象都根据其Property属性（要设置的属性名称）和Value属性（要赋给属性的值），来设置一个属性的值。可将Property中使用的名称完全限定为控件类型（例如Button.Foreground），也可设置Style对象的TargetType属性（例如

Button)，以便解析属性名称。

下面的代码展示了如何使用Style对象来设置Button控件的Foreground属性：

```
<Button>
    Click me!
<Button.Style>
    <Style TargetType="Button">
        <Setter Property="Foreground">
            <Setter.Value>
                <SolidColorBrush Color="Purple" />
            </Setter.Value>
        </Setter>
    </Style>
</Button.Style>
</Button>
```

显然，对于上述代码，用通常方式设置Button控件的Foreground属性会简单得多。将样式转变为资源时，样式就会非常有用，因为资源可供重复使用。稍后将介绍具体用法。

15.2.2 模板

控件用模板构建，而模板可以自定义。模板由一系列控件组成，这些控件按层次结构组合起来，构成了我们看到的控件，其中可能包含用于呈现内容的控件，例如显示内容的按钮。

控件的模板保存在Template属性中，而Template属性是ControlTemplate类的成员。ControlTemplate类包含TargetType属性，该属性可以设置为用于定义模板的控件类型，也可以只包含单个控件。这种控件可以是Grid这样的容器，所以在使用上没有什么限制。

通常，通过样式为类设置模板。方法是按以下方式在Template属性中提供要使用的控件：

```
<Button>
    Click me!
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="Template">

                <Setter.Value>
                    <ControlTemplate TargetType="Button">

                        ...

                    </ControlTemplate>

                </Setter.Value>
```

```
        </Setter>
    </Style>
</Button.Style>
</Button>
```

某些控件可能需要多个模板。例如，CheckBox控件为复选框使用一个模板（CheckBox.Template），为复选框旁的输出文本使用另一个模板（CheckBox.ContentTemplate）。

需要呈现内容的模板都可以在需要输出内容的位置包含一个ContentPresenter控件。某些控件（尤其是需要输出一组项的控件）可以使用其他技术，本章不介绍这样的技术。

替换模板在与资源结合起来时十分有用。但是，由于为控件指定样式是一种极其常用的技术，因此下面用一个“试一试”练习来了解具体实现方式。

试一试：使用样式和模板：Ch15Ex02\MainWindow.xaml

- （1）新建一个WPF应用程序ControlStyling。
- （2）修改MainWindow.xaml中的代码，如下所示：

```
<Grid Background="Black">
```

```
<Button Margin="20" Click="Button_Click">
```

Would anyone use a button like this?

```
<Button.Style>
```

```
<Style TargetType="Button">
```

```
<Setter Property="FontSize" Value="18" />
```

```
<Setter Property="FontFamily" Value="arial" />
```

```
<Setter Property="FontWeight" Value="bold" />
```



```
<Setter Property="Foreground">
```

```
<Setter.Value>
```

```
<LinearGradientBrush StartPoint="0.5,0" EndPoint
```

```
<LinearGradientBrush.GradientStops>
```

```
<GradientStop Offset="0.0" Color="Purple" />
```

```
<GradientStop Offset="0.5" Color="Azure" />
```

```
<GradientStop Offset="1.0" Color="Purple" />
```

</LinearGradientBrush.GradientStops>

</LinearGradientBrush>

</Setter.Value>

</Setter>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Button">

```
<Grid>
```

```
<Grid.ColumnDefinitions>
```

```
<ColumnDefinition Width="50" />
```

```
<ColumnDefinition />
```

```
<ColumnDefinition Width="50" />
```

```
</Grid.ColumnDefinitions>
```

<Grid.RowDefinitions>

<RowDefinition MinHeight="50" />

</Grid.RowDefinitions>

<Ellipse Grid.Column="0" Height="50">

<Ellipse.Fill>

<RadialGradientBrush>

<RadialGradientBrush.GradientStops>

<GradientStop Offset="0.0" Color="Yellow"

<GradientStop Offset="1.0" Color="Red"

</RadialGradientBrush.GradientStops>

</RadialGradientBrush>

</Ellipse.Fill>

</Ellipse>

<Grid Grid.Column="1">

<Rectangle RadiusX="10" RadiusY="10">

<Rectangle.Fill>

<RadialGradientBrush>

<RadialGradientBrush.GradientStops>

<GradientStop Offset="0.0" Color="Yellow">

<GradientStop Offset="1.0" Color="Red">

</RadialGradientBrush.GradientStops>

</RadialGradientBrush>

</Rectangle.Fill>

</Rectangle>

<ContentPresenter Margin="20,0,20,0"

HorizontalAlignment="Center"

VerticalAlignment="Center" />

```
</Grid>
```

```
<Ellipse Grid.Column="2" Height="50">
```

```
<Ellipse.Fill>
```

```
<RadialGradientBrush>
```

```
<RadialGradientBrush.GradientStops>
```

```
<GradientStop Offset="0.0" Color="Yellow">
```

```
<GradientStop Offset="1.0" Color="Red">
```


</RadialGradientBrush.GradientStops>

</RadialGradientBrush>

</Ellipse.Fill>

</Ellipse>

</Grid>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</Button.Style>

</Button>

</Grid>

(3) 修改MainWindow.xaml.cs中的代码，如下所示：

```
public partial class MainWindow : Window
{
    ...
    private void Button_Click(object sender, RoutedEventArgs e
```

```
{  
  
    MessageBox.Show("Button clicked.");  
  
}  
}
```

(4) 运行该应用程序，单击其中的按钮。图15-2呈现了结果。

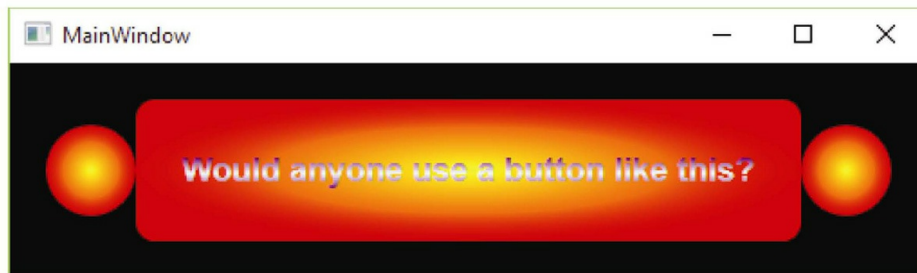


图15-2

示例的说明

首先，这个例子中的按钮十分丑陋。不过，抛开美观方面的考虑，本例说明了，在WF

注意，标准Windows按钮的某些效果在此处使用的模板中并没有实现。特别是，鼠标

在介绍触发器之前，先仔细看一下示例代码，特别注意样式和模板，了解模板是如何

本例的开头是用于显示Button控件的常用代码：

```
<Button Margin="20" Click="Button_Click">
    Would anyone use a button like this?
```

这指定了按钮的基本属性和内容。Style属性设置为一个Style对象，并为Button挂

```
<Button.Style>
  <Style TargetType="Button">
    <Setter Property="FontSize" Value="18" />
    <Setter Property="FontFamily" Value="arial" />
    <Setter Property="FontWeight" Value="bold" />
```

接下来，使用属性元素语法来设置Button.Foreground属性，因为它使用了画笔：

```
<Setter Property="Foreground">
  <Setter.Value>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0.0" Color="Purple" />
        <GradientStop Offset="0.5" Color="Azure" />
        <GradientStop Offset="1.0" Color="Purple" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Setter.Value>
</Setter>
```

Style对象的其余代码将Button.Template属性设置为一个ControlTemplate对

```
<Setter Property="Template">
```

```

        <Setter.Value>
            <ControlTemplate TargetType="Button">
                ...
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
</Button.Style>
</Button>

```

模板代码可以归结为一个Grid控件，它包含1行3列。3个单元格依次包含了一个E11

```

<Grid>
    <Ellipse Grid.Column="0" Height="50">
        ...
    </Ellipse>
    <Grid Grid.Column="1">
        <Rectangle RadiusX="10" RadiusY="10">
            ...
        </Rectangle>
        <ContentPresenter Margin="20,0,20,0"
            HorizontalAlignment="Center"
            VerticalAlignment="Center" />
    </Grid>
    <Ellipse Grid.Column="2" Height="50">

```

```
...
</Ellipse>
</Grid>
```

15.2.3 值转换器

以前用过一些赋值方法，例如将字符串`true`赋给布尔属性。`C#`是类型安全的，编译器

`WPF`内置的转换器只支持标准转换，例如可将`int`转换为`string`，将`bool`转换为`int`

一个十分常见的例子是：逆向布尔转换器。比如，在一个对话框中有一个复选框，对

1. `IValueConverter`接口

要创建值转换器，必须实现`IValueConverter`接口。该接口有两个方法：`Convert`

```
object Convert(object value, Type targetType,
```

```
object parameter, CultureInfo culture);  
object ConvertBack(object value, Type targetType,  
    object parameter, CultureInfo culture);
```

通过Convert方法将某个值转换为目标类型，而ConvertBack方法与其相反。这两

2. ValueConversion特性

除实现上述接口外，还可以在这个类中设置一个特性，用于实现这一转换器。这不是

试一试：创建值转换器：**KarliCards Gui**

下面的例子以之前创建的KarliCards Gui项目为基础。

(1) 创建一个新类，命名为ReversedBoolConverter。

(2) 按如下方式修改该类。还必须包含System.Windows.Data名称空间:

```
[ValueConversion(typeof(bool), typeof(bool))]
```

```
public class InverseBoolConverter : IValueConverter
```

```
{
```

```
    public object Convert(object value, Type targetType, objec
```

```
        System.Globalization.CultureInfo culture)
```

```
{
```

```
    return !(bool)value;
```

```
}
```

```
public object ConvertBack(object value, Type targetType, o
```

```
System.Globalization.CultureInfo culture)
```

```
{
```

```
return !(bool)value;
```

```
}
```

```
}
```

(3) 打开Options.xaml文件，为该窗口新建一个静态资源：

```
<Window.Resources>  
    <local:InverseBoolConverter x:Key="inverseBool" />  
</Window.Resources>
```

(4) 将ComboBox的IsEnabled属性设置为如下绑定：

```
IsEnabled="{Binding ElementName=playAgainstComputerCheck,  
    Path=IsChecked, Converter={StaticResource inverseBool}}"
```

示例的说明

本例实现的这个转换非常简单——仅将true值转换为false，反之亦然。稍后将介绍

在XAML代码中，为转换器建立了一个资源，以便在需要它的绑定中引用它。在绑定中

15.2.4 触发器

WPF中的事件几乎无所不包，例如按钮单击、应用程序启动和关闭事件等。实际上，\

在WPF中，只有一部分类继承自`TriggerAction`，但可以定义自己的类。例如，可作

每个控件都有`Triggers`属性，它可用于直接在该控件上定义触发器。还可以沿着层；

触发器对象的配置如下：

- 要定义`Trigger`对象监视的属性，应使用`Trigger.Property`属性。
- 要定义何时激活`Trigger`对象，应设置`Trigger.Value`属性。
- 要定义`Trigger`触发的操作，应将`Trigger.Setters`属性设置为`Setter`对象的一

这里所指的Setter对象就是前面15.2.1一节介绍的Setter对象。

例如，下面的触发器检查MyBooleanValue属性的值，如果其值为true，触发器就

```
<Trigger Property="MyBooleanValue" Value="true">  
  <Setter Property="Opacity" Value="0.5" />  
</Trigger>
```

其实，这段代码并未包含太多信息，因为它没有与任何控件或样式关联起来。下面的

```
<Style TargetType="Button">  
  <Style.Triggers>  
    <Trigger Property="IsMouseOver" Value="true">  
      <Setter Property="Foreground" Value="Yellow" />  
    </Trigger>  
  </Style.Triggers>  
</Style>
```

上述代码在Button.IsMouseOver属性为true时，将Button控件的Foreground

还可以借助`ControlTemplate.Triggers`属性来实现更多功能，创建包含触发器的

15.2.5 动画

动画是通过故事板创建的。毫无疑问，定义动画的最好方法就是使用`Expression Blend`

故事板使用`Storyboard`对象来定义，该对象可以包含一个或多个时间线（`Timeline`）。

`Storyboard`对象包含在资源字典中，所以必须通过`x:Key`属性来识别它。

在故事板的时间线中，可让应用程序的任何元素中类型为`Double`、`Point`或`Color`的

这三种类型都有两个相关的时间线控件，这些控件都可以用作`Storyboard`的子元素。

下面首先分析不使用关键帧实现简单动画时间线的方法，然后学习使用关键帧的时间

1. 不含关键帧的时间线

不含关键帧的时间线有DoubleAnimation、PointAnimation和ColorAnimatic

表15-2 时间线属性

属性	说明
Name	时间线的名称，通过这个名称可在其他位置引用该时间线
BeginTime	在故事板触发多长时间后开始执行此时间线
Duration	此时间线持续的时长
AutoReverse	是否要在此时间线播放完毕回到最初状态，并将属性值恢复为初始值。该属性为布尔值
RepeatBehavior	指定时间线要重复多久——整数后加上一个x（例如5x）表示重复一定的次数后停止；设置为Forever则表示，只有整个故事板暂停或停止时，才不再重复时间线
FillBehavior	设置当时间线结束时，如果故事板还没有结束，应该如何处理该时间线。设置为HoldEnd表示让属性保持时间线结束时的值（默认值），设置为Stop表示将属性重置为初始值
SpeedRatio	动画相对于其他属性值的加速度。默认值为1，设置为其他值则表示加速或减速播放动画
From	在动画开始播放时要设置的属性初始值。不指定这个属性，则使用属性的当前值
To	在动画结束播放时要设置的属性最终值。不指定这个属性，则使用属性的当前值

By

让某个属性从当前的值变换到当前值与指定值之和。可以单独使用该属性，也可以将其与**From**属性联合使用

例如，下面的时间线让**Rectangle**控件的**Width**属性和**MyRectangle**控件的**Name**属性

```
<Storyboard x:Key="RectangleExpander">
  <DoubleAnimation Storyboard.TargetName="MyRectangle"
    Storyboard.TargetProperty="Width" Duration="00:00:05"
    From="100" To="200" />
</Storyboard>
```

2. 含关键帧的时间线

含关键帧的时间线包括**DoubleAnimationUsingKeyFrames**、**PointAnimationUsingKeyFrames**和**ColorAnimationUsingKeyFrames**

这三种时间线可以包含任意数量的关键帧，每个关键帧都会使属性值以不同的方式变

- 离散（Discrete）——离散关键帧会导致属性值直接变成新值，没有过渡。
- 线性（Linear）——线性关键帧会导致属性值线性过渡到新值。
- 曲线（Spline）——曲线关键帧会导致属性值根据三次贝塞尔曲线函数以非线性的方

因此有9种关键帧对象：DiscreteDoubleKeyFrame、LinearDoubleKeyFrame、

关键帧类有3个属性与上一节介绍的时间线类相同。4个曲线关键帧类还有一个附加属

表15-3 曲线关键帧类的属性

属性	用法
Name	关键帧的名称，以便在其他位置引用该关键帧
KeyTime	此关键帧位于时间线开始之后多久
Value	当动画播放到该关键帧时，属性值应该过渡到（或直接设置）的值
KeySpline	此属性是包含两个数字的两对值，形式为cp1x, cp1y, cp2x, cp2y，它们定义了变换属性值的三次贝塞尔函数（只有曲线关键帧有该属性）

例如，变换Ellipse的Center属性（该属性为Point类型），就能让这个Ellipse

```
<Storyboard x:Key="EllipseMover">
  <PointAnimationUsingKeyFrames Storyboard.TargetName="MyEllip
    Storyboard.TargetProperty="Center" RepeatBehavior="Forever"
    <LinearPointKeyFrame KeyTime="00:00:00" Value="50,50" />
    <LinearPointKeyFrame KeyTime="00:00:01" Value="100,50" />
    <LinearPointKeyFrame KeyTime="00:00:02" Value="100,100" />
    <LinearPointKeyFrame KeyTime="00:00:03" Value="50,100" />
    <LinearPointKeyFrame KeyTime="00:00:04" Value="50,50" />
  </PointAnimationUsingKeyFrames>
</Storyboard>
```

在XAML代码中，Point值表示为x,y。

15.3 WPF用户控件

WPF提供了一组在许多情况下有效的控件。不过，与所有.NET开发框架一样，WPF也

用户控件常从UserControl派生。这个类提供了WPF控件需要的所有基本功能，并仿

选择Project|Add User Control菜单项，即可在项目中添加用户控件。随后，崩

在项目中添加了用户控件后，就可以在该控件上添加其他控件，在代码隐藏文件中配

在创建用户控件的过程中，最重要的内容就是了解如何实现依赖属性。第14章简要介

实现依赖属性

依赖属性可以添加到所有继承自System.Windows.DependencyObject的类。这

要在某个类中实现依赖属性，应在类的定义代码中添加一个带有public和static修

```
public static DependencyProperty MyStringProperty;
```

将这个属性定义为static（静态）似乎很奇怪，这样就可以为该类的所有实例单独定

添加的这个成员必须通过静态的DependencyProperty.Register()方法来配置：

```
public static DependencyProperty MyStringProperty =  
    DependencyProperty.Register(...);
```

该方法包含3到5个参数，如表15-4所示（表中按照参数的使用顺序罗列，前3个为必

表15-4 Register（）方法的参数

参数	用法
string name	属性的名称

Type propertyType	属性的类型
Type ownerType	包含属性的类的类型
PropertyMetadata typeMetadata	额外的属性设置：属性的默认值，以及在属性变更通知和强制类型转换时用到的回调方法
ValidateValueCallback validateValueCallback	用于验证属性值的回调方法

还有其它方法也可以注册依赖属性，例如RegisterAttached()，可用来实现附加

例如，使用三个参数就可以注册MyStringProperty依赖属性：

```
public class MyClass : DependencyObject
{
    public static DependencyProperty MyStringProperty = Dependence
        "MyString",
        typeof(string),
        typeof(MyClass));
}
```

还可以添加一个.NET属性，用于直接访问依赖属性（它不是必需的，如下所述）。丌

```
public string MyString

{

    get { return (string)GetValue(MyStringProperty); }

    set { SetValue(MyStringProperty, value); }

}
```

其中，`GetValue()`和`SetValue()`方法分别获取和设置`MyStringProperty`（即

如果需要设置属性的元数据，就必须使用继承自`PropertyMetadata`的对象，例如F

表15-5 FrameworkPropertyMetadata构造函数的重载版本

参数类型	用法
object defaultValue	属性的默认值
FrameworkPropertyMetadata-Options flags	该参数是一系列标志的组合（来自FrameworkPropertyMetadataOptions枚举），用于为属性指定额外的元数据。例如，使用AffectsArrange来声明属性的变更可能会影响控件的布局。使窗口的布局引擎在属性发生变化时重新计算控件的布局。有关此处可用的所有选项，请参见MSDN文档
PropertyChangedCallback propertyChangedCallback	属性值更改时要调用的回调方法
CoerceValueCallback coerceValueCallback	强制转换属性值的类型时要调用的回调方法
bool isAnimationProhibited	指定该属性是否可在动画过程中发生变化
UpdateSourceTrigger defaultUpdateSourceTrigger	当属性值进行了数据绑定时，该属性会根据UpdateSourceTrigger枚举值的不同，确定数据源何时更新。默认值为PropertyChanged，表示绑定源会随着属性值更改。并非所有时候都需要这样处理，例如TextBox.Text属性就可以使用LostFocus值（在丢失焦点时更新）。这样可以避免绑定源过早更新。还可以使用Explicit值，指定绑定源仅在被请求时才更新（通过调用继承自DependencyObject的类的UpdateSource()方法）

下面的简单例子演示了如何使用`FrameworkPropertyMetadata`来设置属性的默认

```
public static DependencyProperty MyStringProperty =  
    DependencyProperty.Register(  
        "MyString",  
        typeof(string),  
        typeof(MyClass),  
        new FrameworkPropertyMetadata("Default value"));
```

前面学习了三个回调方法，分别用于属性变更通知、属性强制类型转换和属性值的验

现在，继续开发游戏客户端**Karli Cards**。下面的“试一试”练习将在应用程序中创

试一试：用户控件：**KarliCards Gui.CardControl.xaml**

从之前的试一试练习回到KarliCards Gui项目中。

(1) 本例将用到第13章创建的Ch13CardLib项目，因此把它添加到解决方案中。

(2) 添加对Ch13CardLib项目的引用。具体做法为：在KarliCards Gui项目中

(3) 在KarliCards Gui项目中添加一个新用户控件CardControl，然后修改Ca

```
<UserControl x:Class="KarliCards_Gui.CardControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-com
    xmlns:d="http://schemas.microsoft.com/expression/blend
    xmlns:local="clr-namespace:KarliCards_Gui"
    mc:Ignorable="d"
    Height="154" Width="100" Name ="UserControl">
<UserControl.Resources>
    <local:RankNameConverter x:Key="rankConverter"/>
    <DataTemplate x:Key="SuitTemplate">
        <TextBlock Text="{Binding}"/>
    </DataTemplate>
    <Style TargetType="Image" x:Key="SuitImage">
        <Style.Triggers>
```

```

        <DataTrigger Binding="{Binding ElementName=UserControl,
Value="Club">
            <Setter Property="Source" Value="Images\Clubs.png" />
        </DataTrigger>
        <DataTrigger Binding="{Binding ElementName=UserControl, Pat
Value="Heart">
            <Setter Property="Source" Value="Images\Hearts.png" />
        </DataTrigger>
        <DataTrigger Binding="{Binding ElementName=UserControl, Pat
Value="Diamond">
            <Setter Property="Source" Value="Images\Diamonds.png" />
        </DataTrigger>
        <DataTrigger Binding="{Binding ElementName=UserControl, Pat
Value="Spade">
            <Setter Property="Source" Value="Images\Spades.png" />
        </DataTrigger>
    </Style.Triggers>
</Style>
</UserControl.Resources>
<Grid>
    <Rectangle Stroke="{x:Null}" RadiusX="12.5" RadiusY="12.5">
        <Rectangle.Fill>
            <LinearGradientBrush EndPoint="0.47, -0.167" StartPoint="0
                <GradientStop Color="#FFD1C78F" Offset="0"/>
                <GradientStop Color="#FFFFFFFF" Offset="1"/>
            </LinearGradientBrush>
        </Rectangle.Fill>

```

```

    <Rectangle.Effect>
        <DropShadowEffect Direction="145" BlurRadius="10" ShadowColor="Black" />
    </Rectangle.Effect>
</Rectangle>
<Label x:Name="SuitLabel"
    Content="{Binding Path=Suit, ElementName=UserControl, Mode=OneWay}"
    ContentTemplate="{DynamicResource SuitTemplate}"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Margin="8,51,8,60" />
<Label x:Name="RankLabel" Grid.ZIndex="1"
    Content="{Binding Path=Rank, ElementName=UserControl, Mode=OneWay}"
    Converter={StaticResource ResourceKey=rankConverter}"
    ContentTemplate="{DynamicResource SuitTemplate}"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="8,8,0,0" />
<Label x:Name="RankLabelInverted"
    Content="{Binding Path=Rank, ElementName=UserControl, Mode=OneWay}"
    Converter={StaticResource ResourceKey=rankConverter}"
    ContentTemplate="{DynamicResource SuitTemplate}"
    HorizontalAlignment="Right" VerticalAlignment="Bottom"
    Margin="0,0,8,8" RenderTransformOrigin="0.5,0.5">
    <Label.RenderTransform>
        <RotateTransform Angle="180"/>
    </Label.RenderTransform>
</Label>
<Image Name="TopRightImage" Style="{StaticResource ResourceKey=ImageBrush}"
    Margin="12,12,8,0" HorizontalAlignment="Right" VerticalAlignment="Bottom" />

```

```

Width="18.5" Height="18.5" Stretch="UniformToFill" />
    <Image Name="BottomLeftImage" Style="{StaticResource Resour
Margin="12,0,8,12" HorizontalAlignment="Left" VerticalAlignment:
Width="18.5" Height="18.5" Stretch="UniformToFill"
RenderTransformOrigin="0.5,0.5">
    <Image.RenderTransform>
        <RotateTransform Angle="180" />
    </Image.RenderTransform>
</Image>
<Path Fill="#FFFFFFFF" Stretch="Fill" Stroke="{x:Null}"
Margin="0,0,35.218,-0.077" Data="F1 M110.5,51 L145.16457,51
76.731148 115.63518,132.69684 121.63533,149.34013 133.4529
182.12018 152.15821,195.69803 161.79765,200.07669 L110.5,
200 98,194.40356 98,187.5 L98,63.5 C98,56.596439 103.5964
<Path.OpacityMask>
    <LinearGradientBrush EndPoint="0.957,1.127" StartPoint="0,
        <GradientStop Color="#FF000000" Offset="0"/>
        <GradientStop Color="#00FFFFFF" Offset="1"/>
    </LinearGradientBrush>
</Path.OpacityMask>
</Path>
</Grid>
</UserControl>

```

(4) 在该类中添加三个依赖属性:

```

    public static DependencyProperty SuitProperty = DependencyProperty
        "Suit",
        typeof(Ch13CardLib.Suit),
        typeof(CardControl),
        new PropertyMetadata(Ch13CardLib.Suit.Club,
new PropertyChangedCallback(OnSuitChanged)));
    public static DependencyProperty RankProperty = DependencyProperty
        "Rank",
        typeof(Ch13CardLib.Rank),
        typeof(CardControl),
        new PropertyMetadata(Ch13CardLib.Rank.Ace));
    public static DependencyProperty IsFaceUpProperty = Dependence
"IsFaceUp",
    typeof(bool),
    typeof(CardControl),
    new PropertyMetadata(true, new PropertyChangedCallback(OnIsFa
public bool IsFaceUp
{
    get { return (bool)GetValue(IsFaceUpProperty); }
    set { SetValue(IsFaceUpProperty, value); }
}
public Ch13CardLib.Suit Suit
{
    get { return (Ch13CardLib.Suit)GetValue(SuitProperty); }
    set { SetValue(SuitProperty, value); }
}

```

```

}
public Ch13CardLib.Rank Rank
{
    get { return (Ch13CardLib.Rank)GetValue(RankProperty); }
    set { SetValue(RankProperty, value); }
}

```

(5) 在该类中添加更改事件处理程序：

```

public static void OnSuitChanged(DependencyObject source,
    DependencyPropertyChangedEventArgs args)
{
    var control = source as CardControl;
    control.SetTextColor();
}
private static void OnIsFaceUpChanged(DependencyObject source
    DependencyPropertyChangedEventArgs args)
{
    var control = source as CardControl;
    control.RankLabel.Visibility = control.SuitLabel.Visibility
        control.RankLabelInverted.Visibility =
control.TopRightImage.Visibility =
control.BottomLeftImage.Visibility = control.IsFaceUp ?
Visibility.Visible : Visibility.Hidden;
}

```

(6) 在该类中添加如下属性:

```
private Ch13CardLib.Card _card;
public Ch13CardLib.Card Card
{
    get { return _card; }
    private set { _card = value; Suit = _card.suit; Rank =
}
}
```

(7) 添加如下辅助方法, 以设置文本颜色, 重载取牌的构造函数:

```
public CardControl(Ch13CardLib.Card card)
{
    InitializeComponent();
    Card = card;
}
private void SetTextColor()
{
    var color = (Suit == Ch13CardLib.Suit.Club || Suit == Ch
        new SolidColorBrush(Color.FromRgb(0, 0, 0)) :
        new SolidColorBrush(Color.FromRgb(255, 0, 0));
    RankLabel.Foreground = SuitLabel.Foreground = RankLabelI
```

```
        color;  
    }
```

(8) 在项目中新建一个类，增加一个值转换器，命名为RankNameConverter.cs。

```
using System;  
using System.Windows;  
using System.Windows.Data;  
namespace KarliCards_Gui  
{  
    [ValueConversion(typeof(Ch13CardLib.Rank), typeof(string))]  
    public class RankNameConverter : IValueConverter  
    {  
        public object Convert(object value, Type targetType,  
object parameter, System.Globalization.CultureInfo culture)  
        {  
            int source = (int)value;  
            if (source == 1 || source > 10)  
            {  
                switch (source)  
                {  
                    case 1:  
                        return "Ace";  
                    case 11:  
                        return "Jack";  
                }  
            }  
        }  
    }  
}
```



```

        case 12:
            return "Queen";
        case 13:
            return "King";
        default:
            return DependencyProperty.UnsetValue;
    }
}
else
    return source.ToString();
}

public object ConvertBack(object value, Type targetType,
object parameter, System.Globalization.CultureInfo culture)
{
    return DependencyProperty.UnsetValue;
}
}
}

```

(9) 打开GameClient.xaml.cs代码隐藏文件，修改构造函数，如下所示：

```

public GameClient()
{
    InitializeComponent();
    var position = new Point(15, 15);
}

```

```

for (var i = 0; i<4; i++)
{
    var suit = (Ch13CardLib.Suit)i;
    position.Y = 15;
    for (int rank = 1; rank<14; rank++)
    {
        position.Y += 30;
        var card = new CardControl(new Ch13CardLib.Card((Ch13Car
        card.VerticalAlignment = VerticalAlignment.Top;
        card.HorizontalAlignment = HorizontalAlignment.Left;
        card.Margin = new Thickness(position.X, position.Y, 0, 0
        contentGrid.Children.Add(card);
    }
    position.X += 112;
}
}

```

(10) 运行该应用程序，结果如图15-3所示。



图15-3

示例的说明

本例创建了一个用户控件，它带两个依赖属性，并包含使用该控件的客户端代码。该

Card控件包含的大多数代码类似于本章前面介绍的代码。布局代码没有使用新东西，

Card控件的代码向客户端代码提供了三个依赖属性，即**Suit**、**Rank**和**IsFaceUp**，

稍后介绍这些属性的实现代码。现在只需要知道它们都是第10章建立的**CardLib**项目

三个标签显示了牌面大小和花色。尽管它们与不同属性绑定起来，但有一些共同之处

```
<Label x:Name="SuitLabel"
    Content="{Binding Path=Suit, ElementName=UserControl, Mode=
    ContentTemplate="{DynamicResource SuitTemplate}" Horizontal
    VerticalAlignment="Center" Margin="8,51,8,60" />
```

绑定属性值时，也可以指定如何显示绑定的内容，这是通过数据模板（**data templ**

```

<UserControl.Resources>
    <DataTemplate x:Key="SuitTemplate">
        <TextBlock Text="{Binding}"/>
    </DataTemplate>
</UserControl.Resources>

```

Suit的字符串值用作TextBlock控件的Text属性。这个DataTemplate定义在两个

两个Rank标签在绑定中包含一个值转换器。

```

<Label x:Name="RankLabel" Grid.ZIndex="1"
    Content="{Binding Path=Rank, ElementName=UserControl, Mode=OneWay, Converter={StaticResource ResourceKey=rankConverter}}"
    ContentTemplate="{DynamicResource SuitTemplate}"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="8,8,0,0" />

```

该转换器通过下面的声明包含在UserControl资源中：

```

<local:RankNameConverter x:Key="rankConverter"/>

```

如果删除该值转换器，并不会破坏控件。仍可以看到A、2、3、4枚举值。枚举值的名称

最后需要注意的是RankLabel中的Grid.ZIndex="1"属性。在Grid或Canvas中，

要让上述数据绑定正常工作，必须使用前一节介绍的技术定义三个依赖属性。它们在

```
public static DependencyProperty SuitProperty = DependencyProperty.Register(
    "Suit",
    typeof(CardLib.Suit),
    typeof(CardControl),
    new PropertyMetadata(CardLib.Suit.Club,
new PropertyChangedCallback(OnSuitChanged)));
public static DependencyProperty RankProperty = DependencyProperty.Register(
    "Rank",
    typeof(CardLib.Rank),
    typeof(CardControl),
    new PropertyMetadata(CardLib.Rank.Ace));
public static DependencyProperty IsFaceUpProperty = DependencyProperty.Register(
    "IsFaceUp", typeof(bool),
    typeof(CardControl),
    new PropertyMetadata(true, new PropertyChangedCallback(OnIsFaceUpChanged)));
```

依赖属性使用回调方法来验证其值，**Suit**和**IsFaceUp**属性也为其值的更改准备好了

当**Suit**的值改变时，就调用**OnSuitChanged()**回调方法。该方法将文本的颜色设置

```
public static void OnSuitChanged(DependencyObject source,
    DependencyPropertyChangedEventArgs args)
{
    var control = source as CardControl;
    control.SetTextColor();
}
```

SetTextColor()方法是私有的，但显然**OnSuitChanged()**方法仍可以访问它，因

当**IsFaceUp**改变时，该控件会显示或隐藏用来显示控件当前值的图片和标签。

以上就是对**Card**控件需要了解的内容。**GameClient.xaml.cs**代码隐藏文件中的代

15.4 把所有内容结合起来

这个游戏目前已开发了两个独立的对话框、一个纸牌库和一个主窗口，主窗口提供的

注意：

“视图模型”这个术语来自WPF中一种常见的设计模式：模型-视图-视图模型（MVVM）

15.4.1 重构域模型

如前所述，域模型是描述游戏中所用对象的代码。目前，CardLib项目的下列类描述

- Card

- Deck
- Rank
- Suit


除这些类之外，游戏还需要`Player`和`ComputerPlayer`类，下面就添加它们。还需

这有许多工作，现在就开始吧。

注意：

本例没有使用之前章节中的`CardClient`类，因为控制台应用程序和Windows应用程序

试一试：完成域模型：**KarliCards Gui**



本例继续上一个练习的内容。

(1) 在游戏过程中，每个玩家都会有不同的“状态”。可通过**PlayerState**枚举对其

```
[Serializable]
public enum PlayerState
{
    Inactive,
    Active,
    MustDiscard,
    Winner,
    Loser
}
```

(2) 接下来为玩家新建一些事件。为此，需要一些自定义事件参数，因此添加另一

```
public class PlayerEventArgs : EventArgs
{
    public Player Player { get; set; }
```

```

    public PlayerState State { get; set; }
}

```

(3) 还需要为纸牌新建一些事件，所以新建一个类CardEventArgs:

```

public class CardEventArgs : EventArgs
{
    public Card Card { get; set; }
}

```

(4) 枚举ComputerSkillLevel现在位于Karli Cards Gui项目的GameOptions

```

using Ch13CardLib;

```

(5) 也需要修改Deck类。这里不是多次返回本章前面创建的这个类，而是列出其完

```

using System;
using System.Collections.Generic;
using System.Linq;
namespace Ch13CardLib
{

```

```

public delegate void LastCardDrawnHandler(Deck currentDeck)

public class Deck : ICloneable
{
    public event LastCardDrawnHandler LastCardDrawn;
    private Cards cards = new Cards();
    public Deck()
    {
        InsertAllCards();
    }
    protected Deck(Cards newCards)
    {
        cards = newCards;
    }
    public int CardsInDeck
    {
        get { return cards.Count; }
    }
    public Card GetCard(int cardNum)
    {
        if (cardNum >= 0 && cardNum<= 51)
        {
            if ((cardNum == 51) && (LastCardDrawn != null)) LastCa
            return cards[cardNum];
        }
        else
            throw new CardOutOfRangeExpection(cards.Clone() as Cards),
    }
}

```

```

public void Shuffle()
{
    Cards newDeck = new Cards();
    bool[] assigned = new bool[cards.Count];
    Random sourceGen = new Random();
    for (int i = 0; i < cards.Count; i++)
    {
        int sourceCard = 0;
        bool foundCard = false;
        while (foundCard == false)
        {
            sourceCard = sourceGen.Next(cards.Count);
            if (assigned[sourceCard] == false)
                foundCard = true;
        }
        assigned[sourceCard] = true;
        newDeck.Add(cards[sourceCard]);
    }
    newDeck.CopyTo(cards);
}

public void ReshuffleDiscarded(List<Card> cardsInPlay)
{
    InsertAllCards(cardsInPlay);
    Shuffle();
}

public Card Draw()
{

```

```

        if (cards.Count == 0) return null;
        var card = cards[0];
        cards.RemoveAt(0);
        return card;
    }

    public Card SelectCardOfSpecificSuit(Suit suit)
    {
        Card selectedCard = cards.FirstOrDefault(card => card?.suit
            == suit);
        if (selectedCard == null) return Draw();
        cards.Remove(selectedCard);
        return selectedCard;
    }

    public object Clone()
    {
        Deck newDeck = new Deck(cards.Clone() as Cards);
        return newDeck;
    }

    private void InsertAllCards()
    {
        for (int suitVal = 0; suitVal < 4; suitVal++)
        {
            for (int rankVal = 1; rankVal < 14; rankVal++)
            {
                cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
            }
        }
    }
}

```

```

private void InsertAllCards(List<Card> except)
{
    for (int suitVal = 0; suitVal<4; suitVal++)
    {
        for (int rankVal = 1; rankVal<14; rankVal++)
        {
            var card = new Card((Suit)suitVal, (Rank)rankVal);
            if (except?.Contains(card))
                continue;
            cards.Add(card);
        }
    }
}

```

(6) 在游戏中，有两类玩家：**Player**，由真人来操作；**ComputerPlayer**，由游戏

示例的说明

本练习有许多代码，也做了许多修改！不过，在运行应用程序时，却看不到什么变化

用几个新方法扩展了Deck类。当牌堆（Deck）没有牌时，被丢弃的牌应该回收到游

```
public void PerformDraw(Deck deck, Card availableCard)
{
    switch (Skill)
    {
        case ComputerSkillLevel.Dumb:
            DrawCard(deck);
            break;
        default:
            DrawBestCard(deck, availableCard, (Skill == ComputerSkillLevel.Dumb));
            break;
    }
}

public void PerformDiscard(Deck deck)
{
    switch (Skill)
    {
        case ComputerSkillLevel.Dumb:
            int discardIndex = _random.Next(Hand.Count);
            DiscardCard(Hand[discardIndex]);
            break;
        default:
            // Discard the worst card in the hand
            int worstIndex = -1;
            Card worstCard = null;
            for (int i = 0; i < Hand.Count; i++)
            {
                Card card = Hand[i];
                if (card == null) continue;
                if (worstCard == null || card.CompareTo(worstCard) < 0)
                {
                    worstCard = card;
                    worstIndex = i;
                }
            }
            DiscardCard(Hand[worstIndex]);
            break;
    }
}
```



```

        DiscardWorstCard();
        break;
    }
}
private void DrawBestCard(Deck deck, Card availableCard, bool
{
    var bestSuit = CalculateBestSuit();
    if (availableCard.suit == bestSuit)
        AddCard(availableCard);
    else if (cheat == false)
        DrawCard(deck);
    else
        AddCard(deck.SelectCardOfSpecificSuit(bestSuit));
}

```

作弊是让电脑可以根据牌堆中的牌选择特定花色的牌。如果允许电脑作弊，玩家就更

还要注意，Player类实现了INotifyPropertyChanged接口，PlayerName和St

15.4.2 视图模型

视图模型的作用是保存显示它的视图的状态。对于Karli Cards，它表示已经建立了

游戏执行过程的视图模型必须反映游戏中所有的信息。包括以下几个部分：

- 当前玩家从哪个牌堆中摸牌
- 当前玩家可以抽中的牌，而不是去摸牌
- 当前玩家
- 玩家数量

视图模型还能通知各个玩家发生了更改，这也需要再次实现INotifyPropertyChanged

除上述功能外，视图模型还应提供启动新一轮游戏的功能。为此，在菜单中新建一个



试一试：视图模型：KarliCards Gui

本示例继续KarliCards Gui项目。

(1) 在GameOptions类中通过using语句添加如下名称空间：

```
using System.Windows.Input;  
using System.IO;  
using System.Xml.Serialization;
```

(2) 在GameOptions类中添加一个新命令：

```
public static RoutedCommand OptionsCommand = new RoutedComm  
typeof(GameOptions), new InputGestureCollection(new List<Inpu  
{ new KeyGesture(Key.O, ModifierKeys.Control) }));
```

(3) 在该类中添加两个新方法：

```

public void Save()
{
    using (var stream = File.Open("GameOptions.xml", FileMode.C
    {
        var serializer = new XmlSerializer(typeof(GameOptions));
        serializer.Serialize(stream, this);
    }
}
public static GameOptions Create()
{
    if (File.Exists("GameOptions.xml"))
    {
        using (var stream = File.OpenRead("GameOptions.xml"))
        {
            var serializer = new XmlSerializer(typeof(GameOptions));
            return serializer.Deserialize(stream) as GameOptions;
        }
    }
    else
        return new GameOptions();
}

```

(4) 修改Options.xaml.cs代码隐藏文件中的OK单击事件处理程序，如下所示：

```

private void okButton_Click(object sender, RoutedEventArgs e)

```

```

{
    this.DialogResult = true;
    _gameOptions.Save();
    this.Close();
}

```

(5) 删除构造函数中除InitializeComponent调用之外的所有内容，并关联Data

```

public Options()
{
    _gameOptions = GameOptions.Create();
    DataContext = _gameOptions;
    InitializeComponent();
}

```

(6) 打开StartGame.xaml.cs代码隐藏文件，选择构造函数中的最后4行代码。左

```

private void ChangeListBoxOptions()
{
    if (_gameOptions.PlayAgainstComputer)
        playersListBox.SelectionMode = SelectionMode.Single;
    else
        playersListBox.SelectionMode = SelectionMode.Extended;
}

```

(7) 删除构造函数中除InitializeComponent之外的所有内容，并关联DataCor

```
public StartGame()  
{  
    InitializeComponent();  
    DataContextChanged += StartGame_DataContextChanged;  
}
```

(8) 添加StartGame_DataContextChanged事件处理程序:

```
void StartGame_DataContextChanged(object sender,  
DependencyPropertyChangedEventArgs e)  
{  
    _gameOptions = DataContext as GameOptions;  
    ChangeListBoxOptions();  
}
```

(9) 修改OK按钮的单击事件处理程序:

```
private void okButton_Click(object sender, RoutedEventArgs e)
```

```
{  
    var gameOptions = DataContext as GameOptions;  
  
    gameOptions.SelectedPlayers = new List<string>();  
  
    foreach (string item in playerNamesListBox.SelectedItems)  
  
    {  
  
        gameOptions.SelectedPlayers.Add(item);  
  
    }  
  
    this.DialogResult = true;
```

```
this.Close();
```

```
}
```

(10) 新建一个类，命名为GameViewModel。首先实现INotifyPropertyChanged

```
using Ch13CardLib;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Windows.Input;
namespace KarliCards_Gui
{
    public class GameViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private void OnPropertyChanged(string propertyName) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(prc
    }
}
```


(11) 添加一个用于保存当前玩家的属性。该属性应该使用OnPropertyChanged事件

```
private Player _currentPlayer;
public Player CurrentPlayer
{
    get { return _currentPlayer; }
    set
    {
        _currentPlayer = value;
        OnPropertyChanged(nameof(CurrentPlayer));
    }
}
```

(12) 与CurrentPlayer属性类似，再在该类中添加4个属性及其相关的字段。属性

表15-6 属性名和字段名

类型	属性名	字段名
List<Player>	Players	_players
Card	CurrentAvailableCard	_availableCard

Deck	GameDeck	_deck
bool	GameStarted	_gameStarted

(13) 添加如下私有字段，用于保存游戏选项：

```
private GameOptions _gameOptions;
```

(14) 添加两个路由命令：

```
public static RoutedCommand StartGameCommand =  
new RoutedCommand("Start New Game", typeof(GameViewModel),  
new InputGestureCollection(new List<InputGesture>  
{ new KeyGesture(Key.N, ModifierKeys.Control) }));  
public static RoutedCommand ShowAboutCommand =  
new RoutedCommand("Show About Dialog", typeof(GameViewModel));
```

(15) 添加一个新的默认构造函数：

```
public GameViewModel()
```

```

{
    _players = new List<Player>();
    _gameOptions = GameOptions.Create();
}

```

(16) 在游戏开始时，需要对玩家和牌堆进行初始化。在类中添加如下代码：

```

public void StartNewGame()
{
    if (_gameOptions.SelectedPlayers.Count<1 ||
(_gameOptions.SelectedPlayers.Count == 1
&& !_gameOptions.PlayAgainstComputer))
        return;
    CreateGameDeck();
    CreatePlayers();
    InitializeGame();
    GameStarted = true;
}
private void InitializeGame()
{
    AssignCurrentPlayer(0);
    CurrentAvailableCard = GameDeck.Draw();
}
private void AssignCurrentPlayer(int index)
{

```

```

    CurrentPlayer = Players[index];
    if (!Players.Any(x => x.State == PlayerState.Winner))
        Players.ForEach(x => x.State = (x == Players[index] ? PlayerState.Inactive));
}

private void InitializePlayer(Player player)
{
    player.DrawNewHand(GameDeck);
    player.OnCardDiscarded += player_OnCardDiscarded;
    player.OnPlayerHasWon += player_OnPlayerHasWon;
    Players.Add(player);
}

private void CreateGameDeck()
{
    GameDeck = new Deck();
    GameDeck.Shuffle();
}

private void CreatePlayers()
{
    Players.Clear();
    for (var i = 0; i<_gameOptions.NumberOfPlayers; i++)
    {
        if (i<_gameOptions.SelectedPlayers.Count)
            InitializePlayer(new Player { Index = i, PlayerName =
_gameOptions.SelectedPlayers[i] });
        else
            InitializePlayer(new ComputerPlayer { Index = i, Skill =

```

```

_gameOptions.ComputerSkill });
    }
}

```

(17) 最后，为玩家生成的事件添加两个事件处理程序：

```

void player_OnPlayerHasWon(object sender, PlayerEventArgs e)
{
    Players.ForEach(x => x.State = (x == e.Player ? PlayerState
                                PlayerState.Loser));
}
void player_OnCardDiscarded(object sender, CardEventArgs e)
{
    CurrentAvailableCard = e.Card;
    var nextIndex = CurrentPlayer.Index + 1 >= _gameOptions.Num
                                CurrentPlayer.Index + 1;
    if (GameDeck.CardsInDeck == 0)
    {
        var cardsInPlay = new List<Card>();
        foreach (var player in Players)
            cardsInPlay.AddRange(player.GetCards());
        cardsInPlay.Add(CurrentAvailableCard);
        GameDeck.ReshuffleDiscarded(cardsInPlay);
    }
    AssignCurrentPlayer(nextIndex);
}

```

```
}
```

(18) 打开GameClient.xaml文件，在Window声明中添加一个新的名称空间：

```
xmlns:vm="clr-namespace:KarliCards_Gui.ViewModel"
```

(19) 在Window声明的下方，添加一个DataContext声明：

```
<Window.DataContext >  
    <local:GameViewModel />  
</Window.DataContext>
```

(20) 在CommandBindings声明中添加三个命令绑定：

```
    <CommandBinding Command="local:GameViewModel.StartGameComm  
CanExecute="CommandCanExecute" Executed="CommandExecuted" />  
    <CommandBinding Command="local:GameViewModel.ShowAboutCommand"  
CanExecute="CommandCanExecute" Executed="CommandExecuted" />  
    <CommandBinding Command="local:GameOptions.OptionsCommand"  
CanExecute="CommandCanExecute" Executed="CommandExecuted" />
```

(21) 在New Game菜单中添加一个命令:

```
<MenuItem Header="_New Game..." Foreground="Black" Width="200"  
    Command="local:GameViewModel.StartGameCommand" />
```

(22) 在Options菜单项中添加一个命令, 并将Width属性设置为200:

```
<MenuItem Header="_Options" HorizontalAlignment="Left" Width="200"  
    Foreground="Black" Command="local:GameOptions.OptionsCommand" />
```

(23) 在About菜单项中添加一个命令:

```
<MenuItem Header="_About" HorizontalAlignment="Left" Width="140"  
    Foreground="Black" Command="local:GameViewModel.ShowAboutComma
```

(24) 打开代码隐藏文件, 修改CommandCanExecute和CommandExecuted方法,

```
private void CommandCanExecute(object sender, CanExecuteRoute
```

```

{
    if (e.Command == ApplicationCommands.Close)
        e.CanExecute = true;
    if (e.Command == ApplicationCommands.Save)
        e.CanExecute = false;
    if (e.Command == GameViewModel.StartGameCommand)
        e.CanExecute = true;
    if (e.Command == GameOptions.OptionsCommand)
        e.CanExecute = true;
    if (e.Command == GameViewModel.ShowAboutCommand)
        e.CanExecute = true;
    e.Handled = true;
}

private void CommandExecuted(object sender, ExecutedRoutedEvent
{
    if (e.Command == ApplicationCommands.Close)
        this.Close();
    if (e.Command == GameViewModel.StartGameCommand)
    {
        var model = new GameViewModel();
        StartGame startGameDialog = new StartGame();
        var options = GameOptions.Create();
        startGameDialog.DataContext = options;
        var result = startGameDialog.ShowDialog();
        if (result.HasValue && result.Value == true)
        {
            options.Save();
        }
    }
}

```



```

        model.StartNewGame();
        DataContext = model;
    }
}
if (e.Command == GameOptions.OptionsCommand)
{
    var dialog = new Options();
    var result = dialog.ShowDialog();
    if (result.HasValue && result.Value == true)
        DataContext = new GameViewModel(); // Clear current game
}
if (e.Command == GameViewModel.ShowAboutCommand)
{
    var dialog = new About();
    dialog.ShowDialog();
}
e.Handled = true;
}

```

(25) 在构造函数中删除InitializeComponent()调用之外的所有内容。

示例的说明

本例对代码进行了许多修改，并且在运行应用程序时也看不到什么变化，但菜单有变

```
public static RoutedCommand OptionsCommand = new RoutedComm
typeof(GameOptions), new InputGestureCollection(new List<Input
{ new KeyGesture(Key.O, ModifierKeys.Control) }));
public static RoutedCommand StartGameCommand =
new RoutedCommand("Start New Game", typeof(GameViewModel),
new InputGestureCollection(new List<InputGesture>
{ new KeyGesture(Key.N, ModifierKeys.Control) }));
```

将InputGestures列表指派给该命令时，快捷键会自动与菜单项关联起来。

在游戏客户端的代码隐藏文件中，还添加了代码，让两个窗口显示为对话框：

```
if (e.Command == GameViewModel.StartGameCommand)
{
    var model = new GameViewModel();
    StartGame startGameDialog = new StartGame();
    startGameDialog.DataContext = model.GameOptions;
    var result = startGameDialog.ShowDialog();
```

```

    if (result.HasValue && result.Value == true)
    {
        model.GameOptions.Save();
        model.StartNewGame();
        DataContext = model;
    }
}

```

将窗口显示为对话框，就可以返回一个值，表示是否应使用对话框的结果。不能从窗

```

private void okButton_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
    this.Close();
}

```

第14章提到，如果对现有对象实例设置DataContext，就必须通过代码来实现。之

```

<Window.DataContext >
    <vm:GameViewModel />
</Window.DataContext>

```

这个实例确保视图中有一个DataContext，但在与StartGame命令中的新示例做交

GameViewModel包含许多代码，但大多数只是玩家和Desk实例的属性和实例化。

游戏开始后，玩家的状态和GameViewModel使游戏在电脑和玩家做出选择后向前推

```
void player_OnPlayerHasWon(object sender, PlayerEventArgs e)
{
    Players.ForEach(x => x.State = (x == e.Player ? PlayerState.Winner : PlayerState.Loser));
}
```

为玩家创建的其他事件也会在这里处理：CardDiscarded用于表明某个玩家完成了

```
void player_OnCardDiscarded(object sender, CardEventArgs e)
{
    CurrentAvailableCard = e.Card;
    var nextIndex = CurrentPlayer.Index + 1 >= _gameOptions.NumberOfPlayers ?
        CurrentPlayer.Index + 1;
    if (GameDeck.CardsInDeck == 0)
    {
        var cardsInPlay = new List<Card>();
    }
}
```

```
        foreach (var player in Players)
        {
            cardsInPlay.AddRange(player.GetCards());
            cardsInPlay.Add(CurrentAvailableCard);
            GameDeck.ReshuffleDiscarded(cardsInPlay);
        }
        AssignCurrentPlayer(nextIndex);
    }
}
```

该事件处理程序还会检查牌堆中是否还有牌。如果没有，事件处理程序就收集游戏中

从GameClient.xaml.cs代码隐藏文件的CommandExecuted方法中调用StartGa

15.4.3 大功告成

现在，整个游戏已经编写好了，但还不能玩，因为游戏客户端还没有任何显示。要让

这两个用户控件是CardsInHand和GameDecks，前者用于显示玩家的一手牌，后者

注意：

只需要在编辑器中输入`propdp`，然后按两次`Tab`键，就可以添加依赖属性。

试一试：大功告成：**KarliCards Gui**

本例继续开发KarliCards Gui项目。

(1) 右击GameClient项目，并选择Add|User Control。新建一个用户控件，命

(2) 在Grid中添加Label和Canvas控件：

```
<Grid>
```

```
    <Label Name="PlayerNameLabel" Foreground="White" FontWeight="Bold">
```

```

FontSize="14" >
    <Label.Effect>
        <DropShadowEffect ShadowDepth="5" Opacity="0.5" Direct
    </Label.Effect>
</Label>
    <Canvas Name="CardSurface">
    </Canvas>
</Grid>

```

(3) 打开代码隐藏文件，添加下列using指令：

```

using Ch13CardLib;
using System;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Threading;

```

(4) 我们需要4个依赖属性。输入propdp，然后按下Tab键，即可插入属性模板。7

表15-7 CardsinHandControl控件的依赖属性

类型	名称	所属的类	默认值
Player	Owner	CardsInHandControl	null
GameViewModel	Game	CardsInHandControl	null
PlayerState	PlayerState	CardsInHandControl	PlayerState.Default
Orientation	PlayerOrientation	CardsInHandControl	Orientation.Default

(5) Owner属性需要一个在属性更改时调用的回调方法。可将其指定为PropertyMetadataCallback

```
public static readonly DependencyProperty OwnerProperty =
    DependencyProperty.Register(
        "Owner",
        typeof(Player),
        typeof(CardsInHandControl),
        new PropertyMetadata(null, new PropertyChangedCallback(OnOwnerChanged))
    );
```

(6) 与Owner属性类似，PlayerState和PlayerOrientation属性也应注册一个PropertyMetadataCallback

(7) 添加回调方法:

```
private static void OnOwnerChanged(DependencyObject source,
DependencyPropertyChangedEventArgs e)
{
    var control = source as CardsInHandControl;
    control.RedrawCards();
}

private static void OnPlayerStateChanged(DependencyObject
DependencyPropertyChangedEventArgs e)
{
    var control = source as CardsInHandControl;
    var computerPlayer = control.Owner as ComputerPlayer;
    if (computerPlayer != null)
    {
        if (computerPlayer.State == PlayerState.MustDiscard)
        {
            Thread delayedWorker = new Thread(control.DelayDiscard);
            delayedWorker.Start(new Payload { Deck = control.Game.Deck,
            AvailableCard = control.Game.CurrentAvailableCard, Player = computerPlayer });
        }
        else if (computerPlayer.State == PlayerState.Active)
        {
            Thread delayedWorker = new Thread(control.DelayDraw);
            delayedWorker.Start(new Payload { Deck = control.Game.Deck,
            AvailableCard = control.Game.CurrentAvailableCard, Player = computerPlayer });
        }
    }
}
```

```

        }
    }
    control.RedrawCards();
}

private static void OnPlayerOrientationChanged(DependencyOb
DependencyPropertyChangedEventArgs args)
{
    var control = source as CardsInHandControl;
    control.RedrawCards();
}

```

(8) 回调方法需要一系列辅助方法。首先添加一个私有类和两个方法，这两个方法E

```

private class Payload
{
    public Deck Deck { get; set; }
    public Card AvailableCard { get; set; }
    public ComputerPlayer Player { get; set; }
}

private void DelayDraw(object payload)
{
    Thread.Sleep(1250);
    var data = payload as Payload;
    Dispatcher.Invoke(DispatcherPriority.Normal,
new Action<Deck, Card>(data.Player.PerformDraw), data.Deck, da

```

```

    }
    private void DelayDiscard(object payload)
    {
        Thread.Sleep(1250);
        var data = payload as Payload;
        Dispatcher.Invoke(DispatcherPriority.Normal,
new Action<Deck>(data.Player.PerformDiscard), data.Deck);
    }

```

(9) 添加用来绘制控件的方法:

```

private void RedrawCards()
{
    CardSurface.Children.Clear();
    if (Owner == null)
    {
        PlayerNameLabel.Content = string.Empty;
        return;
    }
    DrawPlayerName();
    DrawCards();
}
private void DrawCards()
{
    bool isFaceup = (Owner.State != PlayerState.Inactive);

```

```

        if (Owner is ComputerPlayer)
            isFaceup = (Owner.State == PlayerState.Loser ||
Owner.State == PlayerState.Winner);
        var cards = Owner.GetCards();
        if (cards == null || cards.Count == 0)
            return;
        for (var i = 0; i < cards.Count; i++)
        {
            var cardControl = new CardControl(cards[i]);
            if (PlayerOrientation == Orientation.Horizontal)
                cardControl.Margin = new Thickness(i * 35, 35, 0, 0);
            else
                cardControl.Margin = new Thickness(5, 35 + i * 30, 0,
cardControl.MouseDoubleClick += cardControl_MouseDoubleC
cardControl.IsFaceUp = isFaceup;
            CardSurface.Children.Add(cardControl);
        }
    }
    private void DrawPlayerName()
    {
        if (Owner.State == PlayerState.Winner || Owner.State == P:
            PlayerNameLabel.Content = Owner.PlayerName +
(Owner.State == PlayerState.Winner ?
" is the WINNER" : " has LOST");
        else
            PlayerNameLabel.Content = Owner.PlayerName;
        var isActivePlayer = (Owner.State == PlayerState.Active |

```

```

Owner.State == PlayerState.MustDiscard);
    PlayerNameLabel.FontSize = isActivePlayer ? 18 : 14;
    PlayerNameLabel.Foreground = isActivePlayer ?
new SolidColorBrush(Colors.Gold) :
new SolidColorBrush(Colors.White);
}

```

(10) 最后，添加玩家双击纸牌时调用的处理程序：

```

private void cardControl_MouseDoubleClick(object sender, Mo
{
    var selectedCard = sender as CardControl;
    if (Owner == null)
        return;
    if (Owner.State == PlayerState.MustDiscard)
        Owner.DiscardCard(selectedCard.Card);
    RedrawCards();
}

```

(11) 按照步骤 (1) 创建另一个用户控件，命名为GameDecksControl。

(12) 删除Grid控件，插入一个Canvas控件：

```
<Canvas Name="controlCanvas" Width="250" />
```

(13) 打开代码隐藏文件，在其中引用下列名称空间：

```
using Ch13CardLib;  
using System.Collections.Generic;  
using System.Linq;  
using System.Windows;  
using System.Windows.Controls;  
using System.Windows.Documents;  
using System.Windows.Input;
```

(14) 按照步骤(4)添加4个依赖属性，相应的值如表15-8所示。

表15-8 GameDecksControl控件的依赖属性

类型	名称	所属的类	默认值
bool	GameStarted	GameDecksControl	false
Player	CurrentPlayer	GameDecksControl	null

Deck	Deck	GameDecksControl	null
Card	AvailableCard	GameDecksControl	null

(15) 上述4个属性都需要在属性更改时调用回调方法。按照第(4)步添加各个回调

(16) 添加回调方法，代码如下：

```

private static void OnGameStarted(DependencyObject source,
DependencyPropertyChangedEventArgs e)
{
    var control = source as GameDecksControl;
    control.DrawDecks();
}

private static void OnPlayerChanged(DependencyObject source,
DependencyPropertyChangedEventArgs e)
{
    var control = source as GameDecksControl;
    if (control.CurrentPlayer == null)
        return;
    control.CurrentPlayer.OnCardDiscarded +=
        control.CurrentPlayer_OnCardDiscarded;
    control.DrawDecks();
}

```

```

    }
    private void CurrentPlayer_OnCardDiscarded(object sender, C
    {
        AvailableCard = e.Card;
        DrawDecks();
    }
    private static void OnDeckChanged(DependencyObject source,
DependencyPropertyChangedEventArgs e)
    {
        var control = source as GameDecksControl;
        control.DrawDecks();
    }
    private static void OnAvailableCardChanged(DependencyObject
DependencyPropertyChangedEventArgs e)
    {
        var control = source as GameDecksControl;
        control.DrawDecks();
    }
}

```

(17) 添加DrawDecks方法:

```

private void DrawDecks()
{
    controlCanvas.Children.Clear();
    if (CurrentPlayer == null || Deck == null || !GameStarted)

```



```

        return;
    List<CardControl> stackedCards = new List<CardControl>();
    for (int i = 0; i<Deck.CardsInDeck; i++)
        stackedCards.Add(new CardControl(Deck.GetCard(i)) { Margin:
new Thickness(150 + (i * 1.25), 25 - (i * 1.25), 0, 0), IsFace
        if (stackedCards.Count > 0)
            stackedCards.Last().MouseDownClick += Deck_MouseDoubleC:
        if (AvailableCard != null)
        {
            var availableCard = new CardControl(AvailableCard) { Marg:
new Thickness(0, 25, 0, 0) };
            availableCard.MouseDownClick += AvailalbleCard_MouseDou:
            controlCanvas.Children.Add(availableCard);
        }
        stackedCards.ForEach(x => controlCanvas.Children.Add(x));
    }

```

(18) 最后为纸牌添加下列事件处理程序:

```

void AvailalbleCard_MouseDoubleClick(object sender, MouseButt
{
    if (CurrentPlayer.State != PlayerState.Active)
        return;
    var control = sender as CardControl;
    CurrentPlayer.AddCard(control.Card);
}

```

```

        AvailableCard = null;
        DrawDecks();
    }
    void Deck_MouseDoubleClick(object sender, MouseButtonEventArgs e)
    {
        if (CurrentPlayer.State != PlayerState.Active)
            return;
        CurrentPlayer.DrawCard(Deck);
        DrawDecks();
    }

```

(19) 回到GameClient.xaml文件，删除Row 2中的Grid控件，插入下列新的DockPanel

```

<DockPanel Grid.Row="2">
    <local:CardsInHandControl x:Name="Player2Hand" DockPanel.Dock="Top"
        Height="380" Game="{Binding}"
        VerticalAlignment="Center" Width="180" PlayerOrientation="Horizontal"
        Owner="{Binding Players[1]}" PlayerState="{Binding Players[1].State}"
    <local:CardsInHandControl x:Name="Player4Hand" DockPanel.Dock="Top"
        Height="380" VerticalAlignment="Center" Width="180" PlayerOrientation="Vertical"
        Owner="{Binding Players[3]}" PlayerState="{Binding Players[3].State}" Game="{Binding}"
    <local:CardsInHandControl x:Name="Player1Hand" DockPanel.Dock="Bottom"
        Height="154" VerticalAlignment="Bottom" Width="180" PlayerOrientation="Horizontal"
        Owner="{Binding Players[0]}" PlayerState="{Binding Players[0].State}" Game="{Binding}"

```

```

        PlayerState="{Binding Players[0].State}" Game="{Binding}
<local:CardsInHandControl x:Name="Player3Hand" DockPanel.Dock=
        HorizontalAlignment="Center" Height="154" VerticalAlignm
        PlayerOrientation="Horizontal" Owner="{Binding Players[2
        PlayerState="{Binding Players[2].State}" Game="{Binding}
<local:GameDecksControl Height="180" x:Name="GameDecks" Deck="
        AvailableCard="{Binding CurrentAvailableCard}"
        CurrentPlayer="{Binding CurrentPlayer}"
        GameStarted="{Binding GameStarted}"/>
</DockPanel>

```

(20) 运行该应用程序。会默认启用ComputerPlayer类，玩家数目会设置为两个。

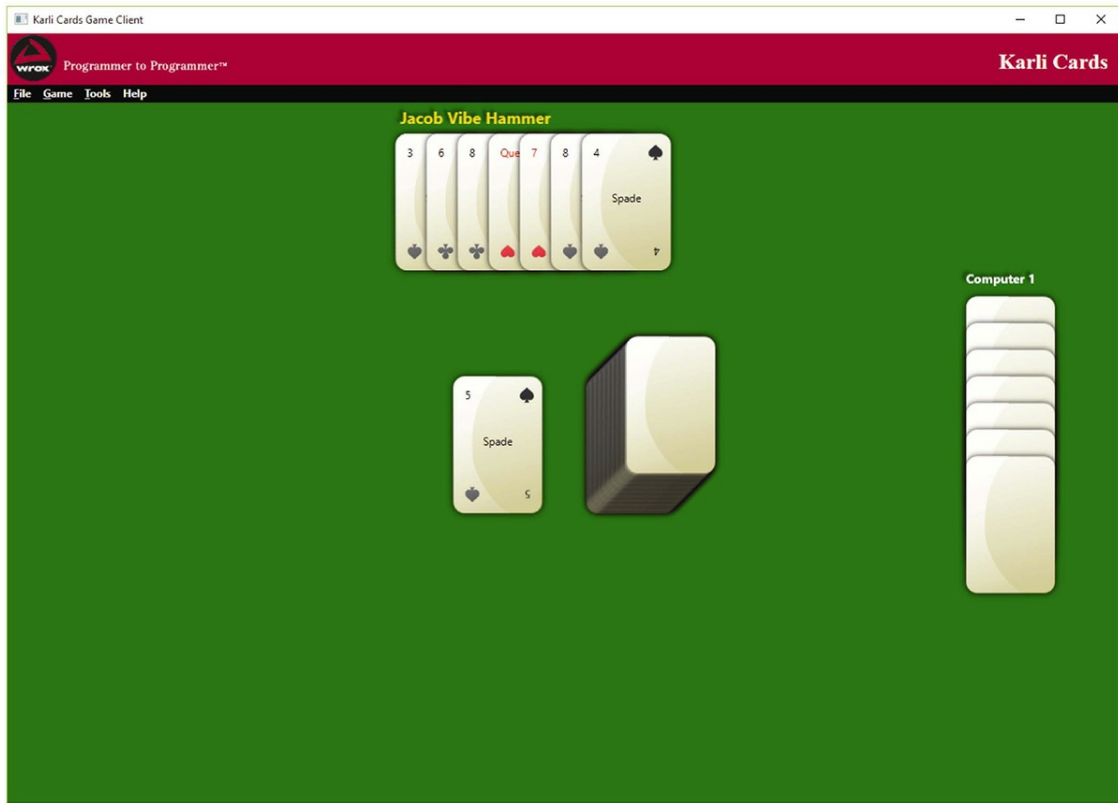


图15-4

双击牌堆或任意可用的牌，即可取牌，然后单击自己手中的一张牌，弃之。

示例的说明

尽管本练习包含了很多代码，但其中大部分都是依赖属性，而XAML都是为这些属性过

```
private static void OnPlayerStateChanged(DependencyObject s
                                         DependencyPropertyChangedEventArgs e)
{
    var control = source as CardsInHandControl;
    var computerPlayer = control.Owner as ComputerPlayer;
    if (computerPlayer != null)
    {
        if (computerPlayer.State == PlayerState.MustDiscard)
        {
            Thread delayedWorker = new Thread(control.DelayDiscard);
            delayedWorker.Start(new Payload
            {
                Deck = control.Game.GameDeck,
                AvailableCard = control.Game.CurrentAvailableCard,
                Player = computerPlayer
            });
        }
        else if (computerPlayer.State == PlayerState.Active)
        {
            Thread delayedWorker = new Thread(control.DelayDraw);
            delayedWorker.Start(new Payload
            {
```

```

        Deck = control.Game.GameDeck,
        AvailableCard = control.Game.CurrentAvailableCard,
        Player = computerPlayer
    });
}
}
control.RedrawCards();
}

```

`OnPlayerStateChanged`方法用来响应玩家状态的变化，判断当前玩家是不是Comp

```

private void DelayDraw(object payload)
{
    Thread.Sleep(1250);
    var data = payload as Payload;
    Dispatcher.Invoke(DispatcherPriority.Normal,
new Action<Deck, Card>(data.Player.PerformDraw), data.Deck, da
    }
}

```

`Dispatcher`用于发起调用，以保证调用是在GUI线程上发生的。

纸牌的绘制很简单。程序会根据`PlayerOrientation`中的设置，将它们水平或垂直

GameDecksControl控件更简单，它使用CurrentPlayer类获得CurrentPlayer

最后在游戏客户端中添加一个DockPanel，在其每一边插入一个CardsInHandControl

```
<local:CardsInHandControl x:Name="Player1Hand" DockPanel.Dock="Top"
    HorizontalAlignment="Center" Height="154" VerticalAlignment="Top"
    PlayerOrientation="Horizontal" Owner="{Binding Players[0]}"
    PlayerState="{Binding Players[0].State}" Game="{Binding Game}" />
```

Game的这个绑定将游戏客户端的DataContext与CardsInHandControl的Game属性

15.5 练习

(1) 这个游戏客户端有一个问题：在**Options**对话框中可以设置电脑玩家的级别，1

提示：此题需要用到**Converter**绑定中的**ConverterParameter**部分。

(2) 电脑可以作弊，玩家当然也希望可以作弊。请在**Options**对话框中添加一个选

(3) 在游戏客户端的底部创建一个状态栏，显示游戏的当前状态。

15.6 本章要点

主题	要点
样式	使用样式，可为XAML元素创建能在许多元素中重复使用的样式。样式允许设置元素的属性。把某个元素的Style属性设置为预定义的样式时，该元素的属性就会使用Style属性中指定的值
模板	模板用于定义控件的内容。通过模板，可以改变标准控件的显示方式，还可以建立复杂的自定义控件
值转换器	值转换器用于在两种类型之间转换值。要创建值转换器，必须让类实现IValueConverter接口
用户控件	用户控件用来创建便于在自己的项目中重复使用的代码和XAML。这些让类代码和XAML也可以导出到其他项目中

第III部分 云编程

- 第16章 基本的云编程
- 第17章 高级云编程和部署

第16章 基本的云编程

本章内容：

- 理解云、云编程和云优化堆栈
- 使用云设计模式为云编程
- 使用Microsoft Azure C#库创建存储容器
- 创建使用存储容器的ASP.NET 4.6网站

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 16 Code后，可找到与本章示例对应的单独文件。

本书中，C#编程的基础知识主要使用控制台应用程序、通过WPF实现的桌面应用程序和Windows通用应用程序来呈现。虽然这些都是引人注目的可行开发技术，但它们不是在云中驻留和运行的好的程序示例。这些程序一般部署、运行在用户的电脑、平板电脑或移动设备上。这些程序编译成可执行文件或动态链接库，依赖.NET Framework等预装软件。所依赖的这些预装软件通常存在于安装上述程序的位置，或者包含在安装过程中。与此相反，在云中运行的互联网应用程序，例如基于

ASP.NET的应用程序，就不能要求访问该程序的计算机或设备上存在任何此类库或依赖的预装软件。所有依赖项都安装在托管互联网应用程序的服务器上，并由设备使用协议（如HTTP、WS（Web Socket）、FTP或SMTP等）来访问。虽然控制台、桌面和Windows通用应用程序可以在云中有依赖的预装软件，如数据库、存储容器或Web服务，但它们自己一般不驻留在云中。

通过Web浏览器访问、响应REST API或WCF服务请求的程序非常适于在云中运行。用于创建这些程序类型的开发技术不需要在调用它们的设备上内置任何所依赖的预装软件。一般情况下，这些程序类型只是彼此交换信息，以清晰、用户友好的方式呈现数据。此外，接收和处理大量数据的程序也非常适合在云中运行，因为利用高可扩展性的资源接受和处理数据是云本身的一个基本特征。

本章将概述什么是云计算，列举在云中成功运行程序的一些模式和技术示例，以及在ASP.NET网站上创建和使用云资源的一个例子。

16.1 云、云编程和云优化堆栈

开始创建完全或部分运行在云上的应用程序只是一个时间问题。它不再是“是否创建”，而是“何时创建”这种应用程序的问题。决定程序的哪些组件运行在云中、云类型和云服务模型，需要调查、理解和计划。对于初学者，必须清楚什么是云。云只是运行在一个数据中心的大量商品化的计算机硬件，这个数据中心可以运行程序，存储大量数据。区别是弹性，即动态向上扩展的能力（例如增加CPU和内存）和/或动态向外扩展的能力（例如增加虚拟服务器实例的数量），而收缩时似乎毫不费力。这与当前的IT运营格局完全不同，在当前的IT运营格局中，被区分开来的计算机资源在公司的一个领域往往会部分或完全未使用，而在其他领域又严重缺乏计算机资源。云解决了这个问题：云可以在需要时提供对计算机资源的访问，在不需要它们时，就将这些资源提供给别人。对于个人开发者，云可以用于部署程序，向外界公布它。如果程序比较受欢迎，就可以扩展它来满足资源需求；如果程序失败了，也不必耗费太多的金钱和时间来建立专用的计算机硬件和基础设施。

下面更详细地探索云类型和云服务模式。常见的云类型有公共云、私人云和混合云，并在以下要点中描述，如图16-1所示。

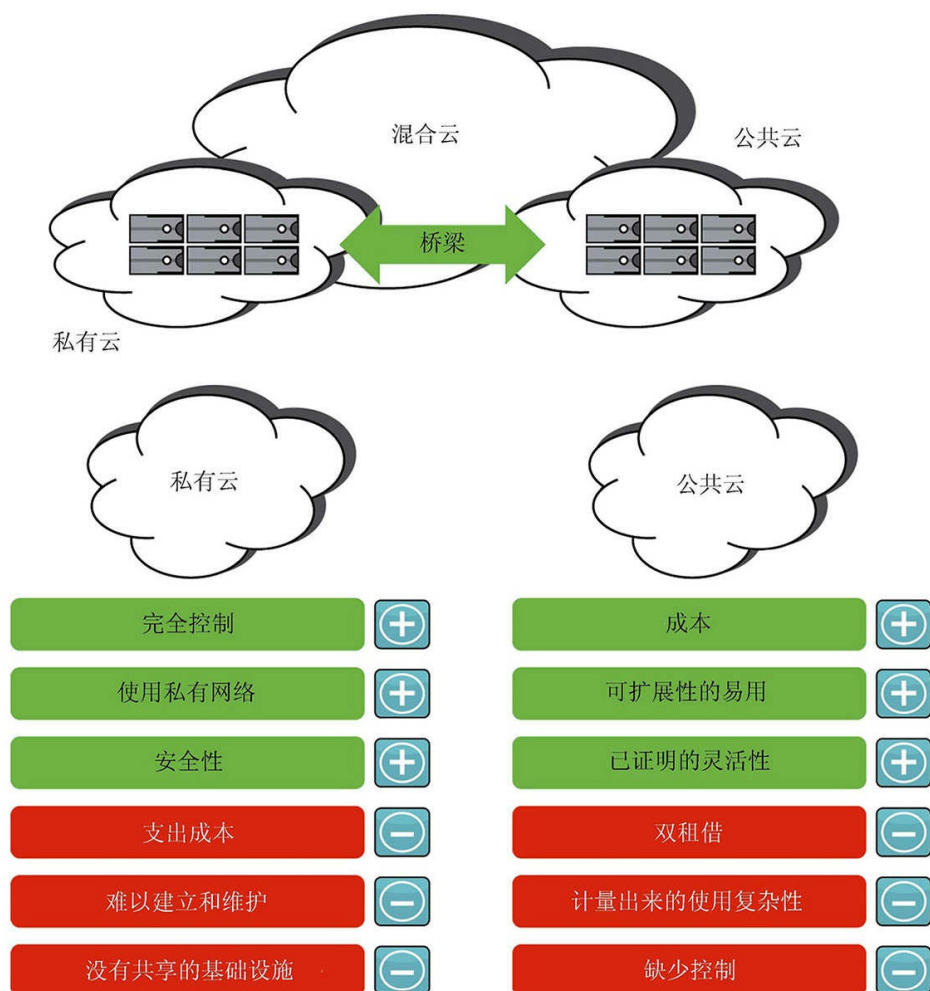


图16-1

- 公共云： 共享云提供者拥有和运营的计算机硬件和基础设施，云提供者有Microsoft Azure、Amazon AWS、Rackspace或IBM Cloud。对于中小企业而言，如果所管理的客户和用户要求不断波动，这种云类型将非常适合。
- 私有云： 这是位于现场或在一个外包数据中心上的专用计算机硬件和基础设施。这种云适合于大公司或必须提供更高级别的数据安全性或服从政府的公司。
- 混合云： 这是公共云和私有云的组合类型，在这种类型中，要选

择IT解决方案的哪些部分在私有云上运行，哪些部分在公共云上运行。理想的解决方案是在私有云上运行对业务至关重要的、需要更高安全级别的程序，在公共云上运行不敏感、可能失效的任务。

云服务模型的数量在不断增加，但最常见的云服务模式如下所示，另见图16-2。

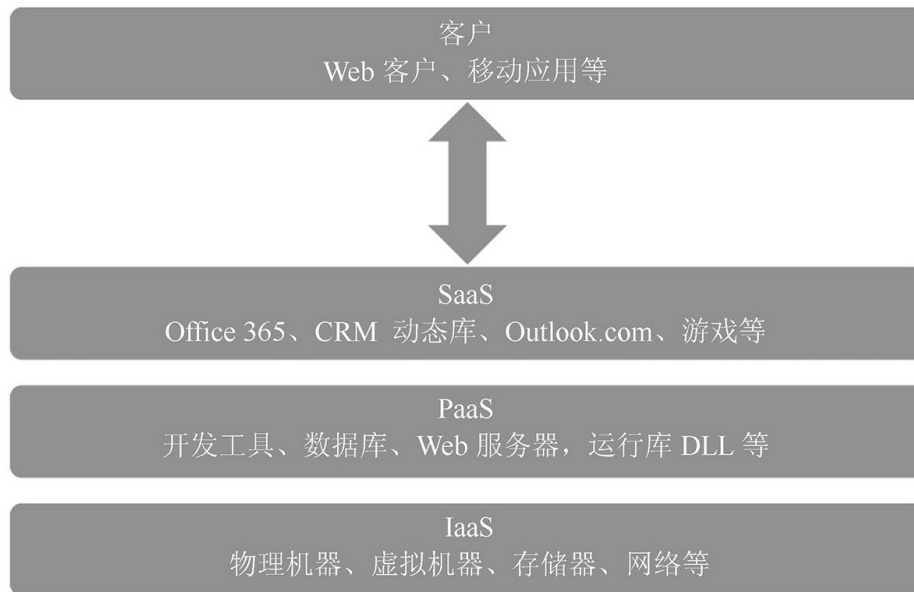


图16-2

- 基础设施即服务（**Infrastructure as a Service, IaaS**）： 要从操作系统开始向上负责。不负责硬件或网络设施；但负责操作系统补丁和第三方依赖库。
- 平台即服务（**Platform as a Service, PaaS**）： 只负责运行在所选操作系统上的程序及其依赖项。不负责操作系统维护、硬件或网络基础架构。
- 软件即服务（**Software as a Service, SaaS**）： 通过互联网访问的设备上使用的一个软件程序或服务。例如，Office 365、

Salesforce、OneDrive或Box，都是通过互联网连接在任意地方访问，并非只有把软件安装在客户端才能起作用。只负责运行在平台上的软件。

总之，云是一个商品化的计算机硬件的弹性结构，用于运行程序。这些程序在混合云、公共云或私有云类型中运行在IaaS、PaaS或SaaS服务模型上。

云编程就是开发运行在任何云服务模型上的代码逻辑。云程序应该包含可移植性、可伸缩性和弹性模式，改善程序的性能和稳定性。没有实现这些可移植性、可伸缩性和弹性模式的程序可以运行在云中，但某些情况下，诸如硬件故障或网络延迟的问题可能导致程序执行意想不到的代码路径，并终止。

注意：云编程模式和最佳实践方式参见下一节。

云的反射弹性是云最大的一个优点，这很重要，因为不仅平台可以扩展，云程序也可以扩展。例如，代码依赖后端资源、数据库、读取或打开文件，还是通过大型数据对象来解析？此类功能操作放在一个云程序中，可以降低其伸缩性，因此支持较低的吞吐量。确保云程序管理的代码路径能执行长期运行的方法，如果需要，把它们放进一个离线处理机制中。

云优化堆栈是一个概念，指代码可以处理高吞吐量，占用空间小，可与其他应用程序一起运行在同一台服务器上，还可以跨平台使用。占用空间小，就可以仅把组件打包到存在依赖项的云程序中，使部署尺寸

尽可能小。云程序需要整个.NET Framework才能起作用？代码不需要整个.NET Framework，而是只包括运行云程序所需要的库，然后把云程序编译到一个自包含的应用程序中以支持并行执行。云程序可与其他任何云程序一起运行，因为它在二进制包中包含了依赖项。最后，使用C#的一个开源版本Mono，云程序就可以打包、编译、部署到Microsoft之外的操作系统上，例如Mac OS X、Linux或UNIX。

16.2 云模式和最佳实践

在云中，增加的延迟或停机时间非常短，代码必须为此做好准备，且包含从这些平台异常中成功恢复的逻辑。如果以前曾现场编码，或编写预先执行的程序代码，这是一个重要的思想转变。需要忘掉很多有关管理异常的东西，学会接受失败，并创建从这种失败中恢复的代码。

上一节中提到了可移植性、可伸缩性和弹性等词，并将这些概念集成到运行在云中的程序里。但这里的可移植性有什么特别含义吗？如果程序可在多个平台上移动或执行，例如Windows、Linux和Mac OS X，该程序就是可移植的。一些ASP.NET 4.6特性位于开源技术的一个新堆栈上，为开发人员提供把代码编译到二进制文件中的选项，以便在这些平台上运行。传统上，开发人员使用ASP.NET编写程序，在后台运行C#，使用IIS在Windows服务器上运行该程序。然而，从以云为核心的角度看，在没有人工干预或程序化干预的情况下，程序及其所有依赖项从一个虚拟机移动到另一个虚拟机的能力，是最适用的可移植性。记住，云中会出现失败，运行程序的虚拟机（VM）可以在任何给定的时间消失，然后在另一个虚拟机上重新构建。因此，程序必须是可移植的，能从这样的事件中恢复。

可伸缩性意味着，当多个客户使用代码时，代码能响应得很好。例如，如果每分钟有1500个请求，且请求的完成和回应应在1秒内完成，则大约每秒有25个并发请求。但是，如果每分钟有15 000个请求，则每秒有250个并发请求。云程序以相同的方式回应25个和250个并发请求吗？2550个并发请求呢？以下是几个有效管理可伸缩性的云编程模式：

- 命令和查询责任隔离（**Command and Query Responsibility Segregation, CQRS**）模式 ——这种模式涉及把读取数据的操作与修改或更新数据的操作分离开。
- 物化视图模式 ——这会修改存储结构，以便反映数据查询模式。例如，为非常常用的查询创建视图可以进行更有效的查询。
- 分片（**Sharding**）模式 ——这把数据分解到多个水平碎片中（其中包含明显不同的数据子集），而不是通过增加硬件的容量，进行垂直伸缩。
- 管家钥匙（**Valet Key**）模式 ——这允许客户直接访问数据存储，以传输或上载大文件。它不是让Web客户机管理数据存储的守卫工作，而是给客户提供一个管家钥匙，并允许直接访问数据存储。

注意： 这些模式涵盖一些先进的C#编码技术，因此这里只描述模式。如果希望看看实现模式的实际C#代码，它们肯定存在，可以在互联网上通过搜索找到它们。

弹性是指程序响应和从服务故障和异常中恢复的程度。从历史上看，IT基础设施一直专注于失败的预防，其可接受的停机时间是最小的，期望值是99.99%或99.999% SLA（**Service-Level Agreement**，服务水平协议）。但是在云中运行程序，可靠性需要一个思维转变，我们需要拥抱失败，要更关注恢复（而不是预防）。程序有多个依赖项，如数据库、存储器、网络和第三方服务，其中一些没有SLA，所以需要这种视角的转变。在出现中断或正常运行未考虑到的情况下，如果仍能做出用户友好的响应，会使云程序富有弹性。下面的一些云编程模式可用于将弹性嵌入云程序：

- 断路器模式 ——这是一种代码设计方式，它了解远程服务的状态，只有服务是可用的，才会试图连接。如果通过以前的失败知道远程服务不可用，就会避免尝试请求，浪费CPU周期。
- 健康端点监控模式 ——这会通过实现端点检测，来检查基于云的应用程序是否可用。
- 重试模式 ——在短暂的异常或故障后重试请求。这种模式在一个给定的时间段内重试多次，重试尝试次数到达阈值时，就停止重试。
- 节流模式 ——管理云程序的使用，以便达到SLA，程序在高负载下仍然可用。

使用上述的一个或多个模式，有助于更成功地实现云迁移。上述模式会提高程序的可伸缩性和弹性，从而提高程序的可用性。这反过来会带来更愉悦的用户或客户体验。

16.3 使用Microsoft Azure C#库创建存储容器

尽管有许多云提供商，用于本章和下一章中例子的云提供商是Microsoft。Microsoft提供的云平台是Azure。Azure有许多不同种类的特性。例如IaaS产品称为Azure VM，PaaS产品称为Azure云服务。此外，Microsoft还有用于数据库的SQL Azure、用于验证用户身份的Azure Active Directory、用于存储blob的Azure Storage。

注意： 下面的“试一试”练习要求订阅Microsoft Azure。如果没有，可以在<http://azure.microsoft.com>上注册一个免费试用版本。

以下两个练习将使用用于.NET和C#的Microsoft Azure Storage Client Library创建一个Azure存储账户，再创建一个Azure存储容器。下一节将使用Visual Studio创建一个ASP.NET 4.6 Web站点，来访问存储在Azure存储容器中的图像。在本书的后面，ASP.NET 4.6 Web站点将处理一手扑克牌。扑克牌图像是存储在Azure存储容器中的blob。

试一试： 创建一个**Azure**存储账户

(1) 访问Microsoft

Azure门户网站

<https://manage.windowsazure.com>。

(2) 单击STORAGE特性，然后单击CREATE A STORAGE ACCOUNT，如图16-3所示。

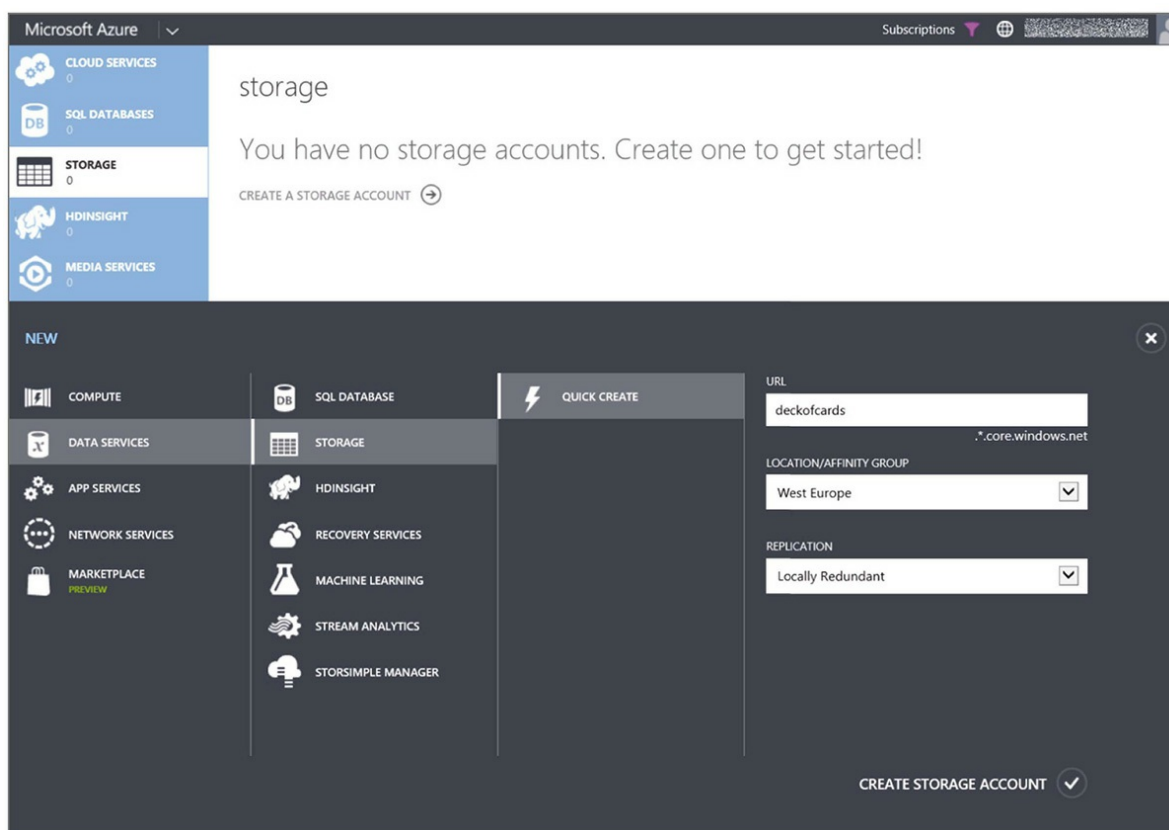


图16-3

(3) 输入存储账户名称、位置和复制数据。由于成本原因，考虑把冗余设置为Locally Redundant。这就避免了在另一个区域创建存储账户的浅度副本而带来的少量额外成本。然而在存储账户所在的同一区域数据中心，文件复制了三次。

(4) 一旦输入名称、位置和复制数据，就单击CREATE STORAGE ACCOUNT并确认看到如图16-4所示的屏幕。在这个例子

中，Azure存储账户是deckofcards。给Azure存储账户指定另一个名称。记住该名字，因为它要用于下一个“试一试”练习。

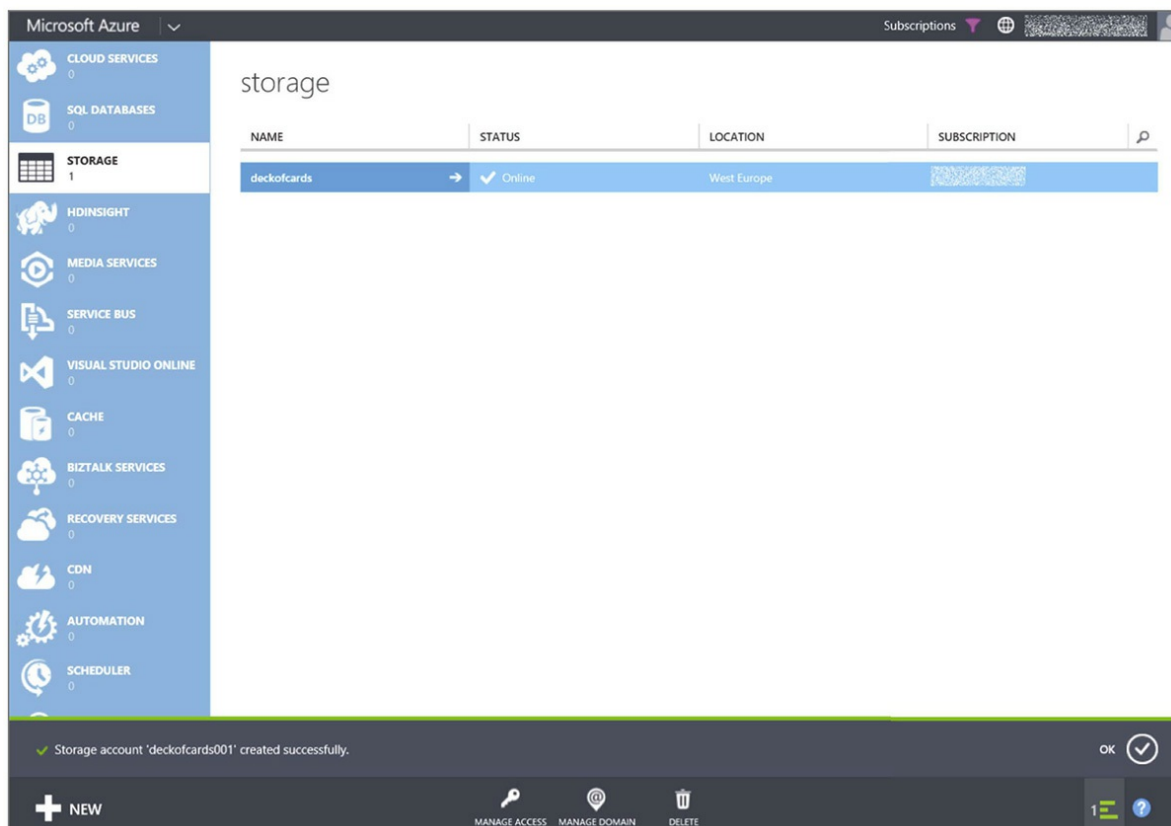


图16-4

(5) 现在就成功地创建了一个Azure存储账户。

注意： 存储账户可用于存储blob、表、队列和文件。例子包括数据库备份、Azure Web App IIS日志、VM机器映像、文档或图片，每个存储账户最多可存储100TB数据。

示例说明

Microsoft Azure管理控制台本身运行在PaaS云服务模式（即Azure云服务）的Microsoft Azure平台上。管理控制台由Microsoft的一支产品团队编写，由Microsoft支持人员支持。左边导航栏上的所有特性都可以创建和利用。用自己的订阅创建一个Azure存储账户，就可以获得存储空间和一个全局可访问的URL，进而访问存储账户的内容（例如 <https://deckofcards.blob.core.windows.net>）。

试一试：使用Microsoft Azure Storage Client Library创建存储容器

下面将使用Visual Studio 2015和Microsoft Azure Storage Client库来创建一个控制台应用程序，再创建一个Azure存储容器，并给它上传52张牌。

（1）在Visual Studio中选择File|New|Project，创建一个新的控制台应用程序项目。在New Project对话框中（参见图16-5），选择类别Visual C#和子类别Windows，然后选择Console Application模板，把项目命名为Ch16Ex01。

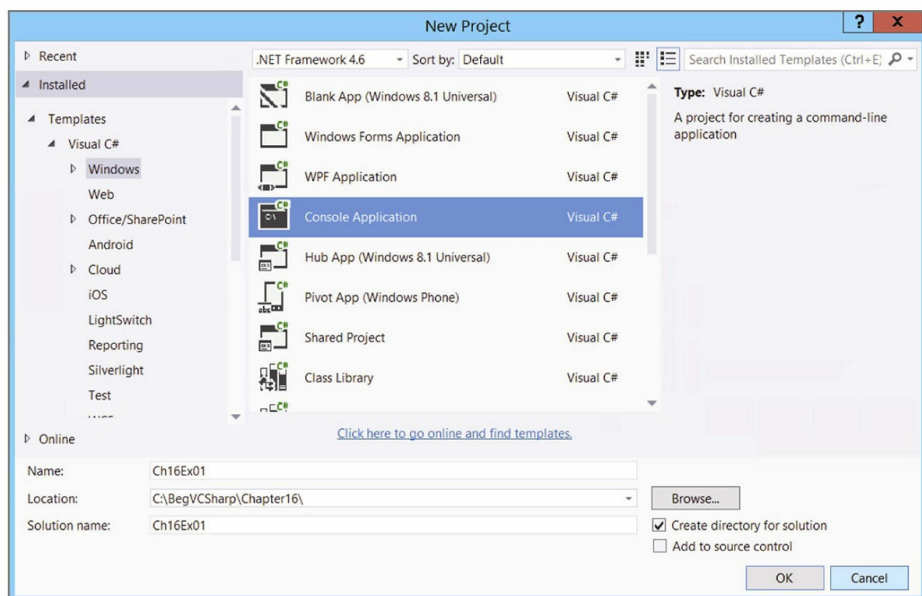


图16-5

(2) 右击Ch16Ex01|Add...|New Folder，给项目添加一个名为Cards的目录。把52张扑克牌图像添加到目录中，如图16-6所示。图像可从源代码下载站点中获得，分别命名，即从0-1.PNG到3-13.PNG。

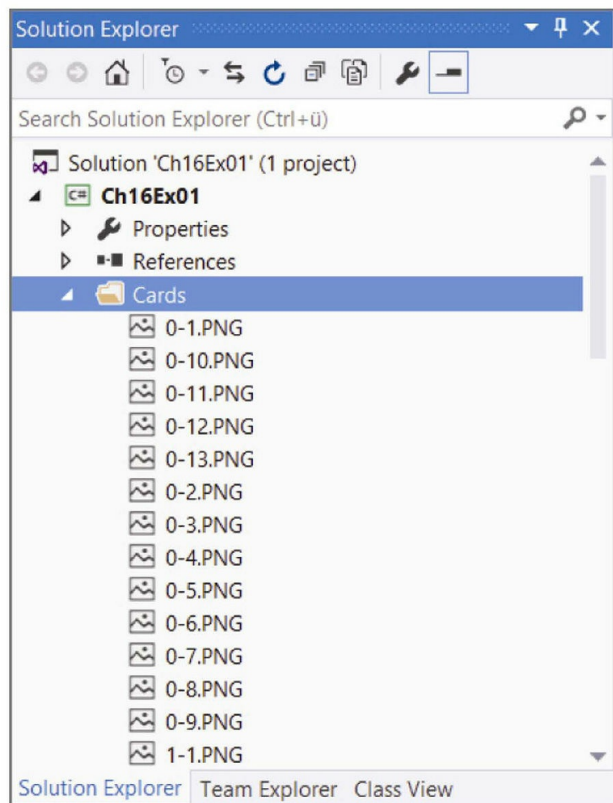


图16-6

(3) 此外，把Cards目录复制到
C:\BegVCSharp\Chapter16\Ch16Ex01\Ch16Ex01 bin\Debug下，这样在运行时，编译后的可执行文件可以找到它们。

(4) 再次右击Ch16Ex01项目，从弹出菜单中选择Manage NuGet Packages...。

(5) 在如图16-7所示的搜索文本框中，输入Windows Azure Storage，安装WindowsAzure.Storage客户端库。Windows Azure Storage库的更多信息可在<https://msdn.microsoft.com/library/azure/dn261237.aspx>中找到。

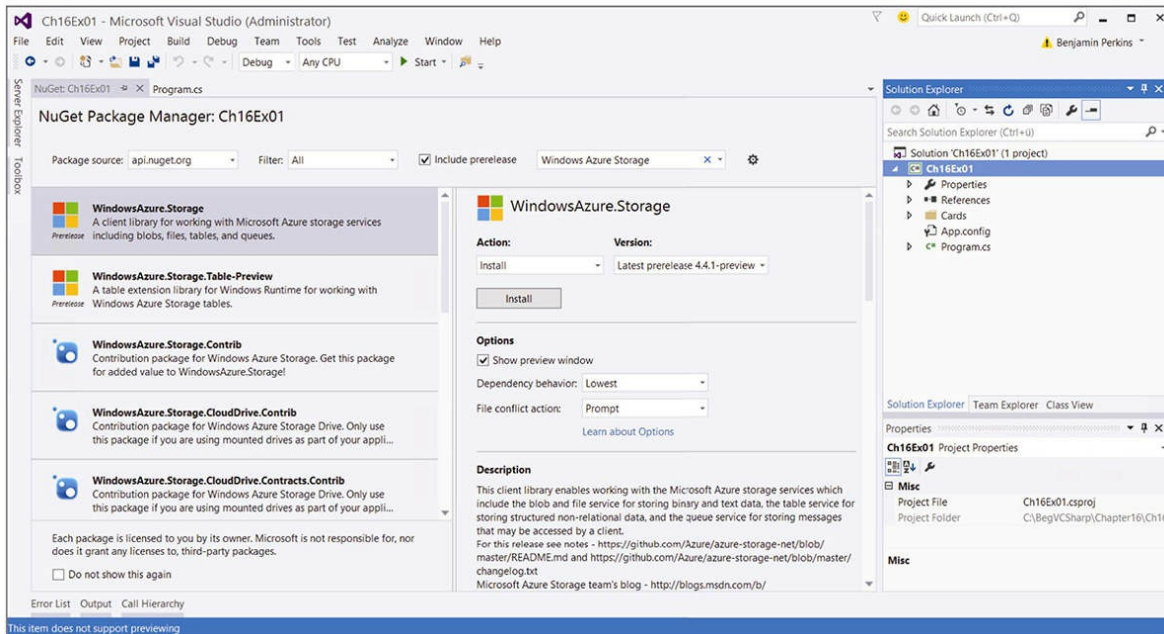


图16-7

（6）接受用户协议，一旦安装NuGet包及其依赖项，就会在Visual Studio的输出窗口中看到“=====Finished=====”消息。此外，Ch16Ex01内的References文件夹展开了，可以查看新添加的二进制文件。

（7）打开App.config文件，将以下<appSetting>设置添加到<configuration>部分。注意在前面的“试一试”练习（deckofcards）中，AccountName是Azure存储账户的名称。应把它改为自己的Azure存储账户名。获得AccountKey的步骤可参考步骤（8）。

```
<appSettings>
  <add key="StorageConnectionString"
    value="DefaultEndpointsProtocol=https;AccountName=<NAME>;
    AccountKey=<KEY>" />
```

</appSettings>

(8) 为获得Azure存储账户键，应访问Microsoft Azure管理门户，导航到Azure存储账户。如图16-8所示，页面底部有一项Manage Access Keys。选择该项，把PRIMARY ACCESS KEY复制到剪贴板上，再复制到App.config文件中，作为AccountKey的值。

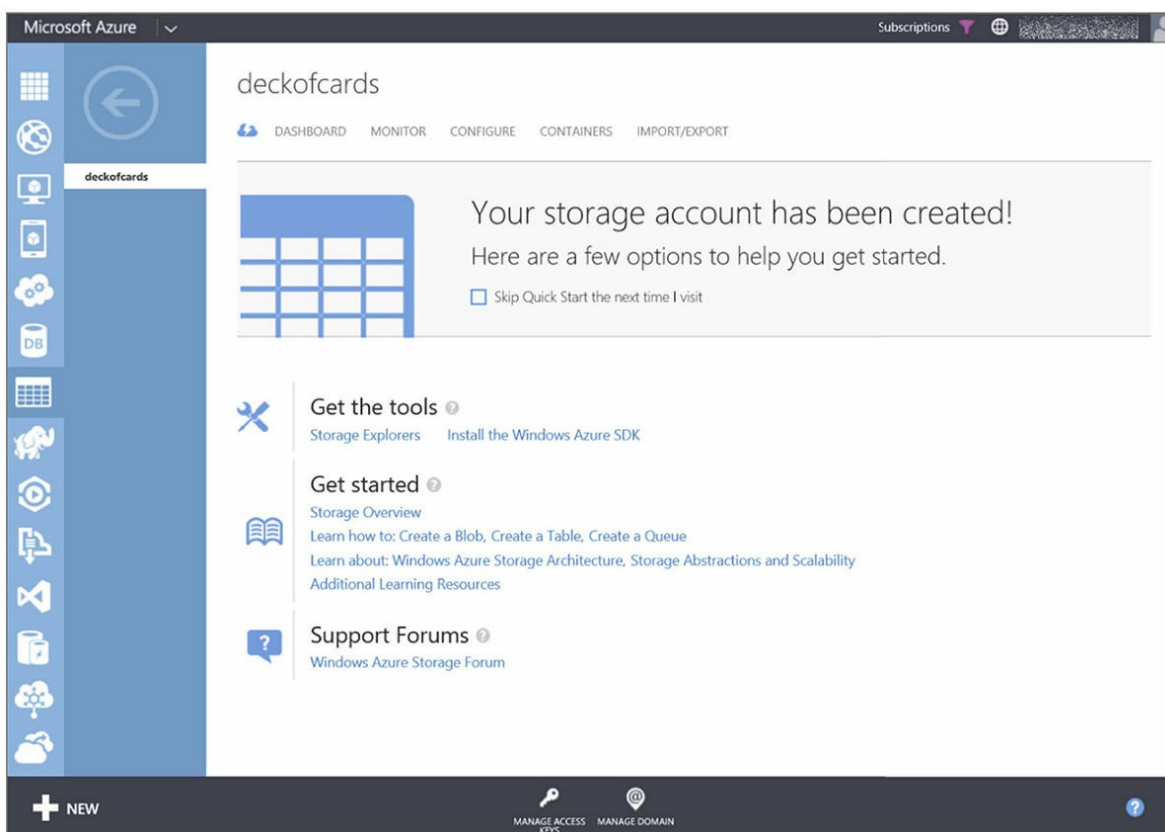


图16-8

(9) 现在添加代码来创建容器，上传图片，列出它们，如有必要，就删除它们。首先在Main()方法中添加程序集引用和C#框架try { } ...catch { }，如下所示：

```
using static System.Console;
```

```

using System.IO;
using Microsoft.WindowsAzure;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Auth;
using Microsoft.WindowsAzure.Storage.Blob;
static void Main(string[] args)
{
    try {
    }
    catch (StorageException ex)
    {
        WriteLine($"StorageException: {ex.Message}");
    }
    catch (Exception ex)
    {
        WriteLine($"Exception: {ex.Message}");
    }
    WriteLine("Press enter to exit.");
    ReadLine();
}

```

(10) 接下来，在try{}代码块中添加创建容器的代码，如下所示。看看传递给blobClient.Get-ContainerReference("carddeck")的参数carddeck。这是用于Azure存储容器的名称。此后可通过[https://deckofcards.blob.core.windows.net/carddeck/0-1. PNG](https://deckofcards.blob.core.windows.net/carddeck/0-1.PNG)访问这个容器的内容。可以在其中放置任何想要的名字，只要符合命名要求即可（例如，名称必须有3到63个字符长，必须以字母或数字开头）。如果提供

的容器名称不符合命名要求，就返回400 HTTP状态错误。

```
CloudStorageAccount storageAccount =  
CloudStorageAccount.Parse(CloudConfigurationManager.GetSetting  
CloudBlobClient blobClient = storageAccount.CreateCloudBlobCli  
CloudBlobContainer container = blobClient.GetContainerReferenc  
if (container.CreateIfNotExists())  
{  
    WriteLine($"Created container '{container.Name}' " +  
        $"in storage account '{storageAccount.Credentials.AccountId}'");  
}  
else  
{  
    WriteLine($"Container '{container.Name}' already exists " +  
        $"for storage account '{storageAccount.Credentials.AccountId}'");  
}  
container.SetPermissions(new BlobContainerPermissions  
{ PublicAccess = BlobContainerPublicAccessType.Blob });  
WriteLine($"Permission for container '{container.Name}' is public");
```

(11) 把如下代码添加到创建容器的代码后面，这些代码会上传存储在Cards文件夹中的扑克牌图像：

```
int numberOfCards = 0;  
DirectoryInfo dir = new DirectoryInfo(@"Cards");  
foreach (FileInfo f in dir.GetFiles("*.png"))  
{  
    CloudBlockBlob blockBlob = container.GetBlockBlobReference(f.Name);  
    blockBlob.UploadFromFile(f.FullName);  
    numberOfCards++;  
}
```

```

using (var fileStream = System.IO.File.OpenRead(@"Cards\" + f
{
    blockBlob.UploadFromStream(fileStream);
    WriteLine($"Uploading: '{f.Name}' which " +
        $"is {fileStream.Length} bytes.");
}
numberOfCards++;
}
WriteLine($"Uploaded {numberOfCards.ToString()} cards.");
WriteLine();

```

(12) 图片上传后，检查一切是否正常。添加下面的代码，列出存储在新建的Azure存储容器carddeck中的blob。

```

numberOfCards = 0;
foreach (IListBlobItem item in container.ListBlobs(null, false
{
    if (item.GetType() == typeof(CloudBlockBlob))
    {
        CloudBlockBlob blob = (CloudBlockBlob)item;
        WriteLine($"Card image url '{blob.Uri}' with length " +
            $" of {blob.Properties.Length}");
    }
    numberOfCards++;
}
WriteLine($"Listed {numberOfCards.ToString()} cards.");

```

(13) 现在，如有必要，可以删除刚刚上传的图片。下面的例子展

示了如何以编程方式删除容器中的blob文件：

```
WriteLine("Enter Y to delete listed cards, press enter to skip  
if (ReadLine() == "Y")  
{  
    numberOfCards = 0;  
    foreach (IListBlobItem item in container.ListBlobs(null, false)  
    {  
        CloudBlockBlob blob = (CloudBlockBlob)item;  
        CloudBlockBlob blobToDelete = container.GetBlockBlobReference(blob.Name);  
        blobToDelete.Delete();  
        WriteLine($"Deleted: '{blob.Name}' which was {blob.Name.Length} bytes");  
        numberOfCards++;  
    }  
    WriteLine($"Deleted {numberOfCards.ToString()} cards.");  
}
```

(14) 运行控制台应用程序并查看输出，结果如图16-9所示。然后访问Microsoft Azure管理控制台，查看新建容器carddeck的页面，如图16-10所示。单击容器，查看其内容。


```
file:///C:/BegVCSharp/Chapter16/Ch16Ex01/Ch16Ex01/bin/...
Created container 'carddeck' in storage account 'deckofcards'.
Permission for container 'carddeck' is public.
Uploading: '0-1.PNG' which is 3932 bytes.
Uploading: '0-10.PNG' which is 9985 bytes.
Uploading: '0-11.PNG' which is 19657 bytes.
Uploading: '0-12.PNG' which is 19143 bytes.
Uploading: '0-13.PNG' which is 19091 bytes.
Uploading: '0-2.PNG' which is 4744 bytes.
Uploading: '0-3.PNG' which is 5621 bytes.
Uploading: '0-4.PNG' which is 5716 bytes.
Uploading: '0-5.PNG' which is 6872 bytes.
Uploading: '0-6.PNG' which is 7713 bytes.
Uploading: '0-7.PNG' which is 8480 bytes.
Uploading: '0-8.PNG' which is 9716 bytes.
Uploading: '0-9.PNG' which is 9722 bytes.
Uploading: '1-1.PNG' which is 3078 bytes.
Uploading: '1-10.PNG' which is 5769 bytes.
Uploading: '1-11.PNG' which is 18692 bytes.
Uploading: '1-12.PNG' which is 19142 bytes.
Uploading: '1-13.PNG' which is 19543 bytes.
Uploading: '1-2.PNG' which is 3109 bytes.
Uploading: '1-3.PNG' which is 3623 bytes.
Uploading: '1-4.PNG' which is 3376 bytes.
Uploading: '1-5.PNG' which is 3925 bytes.
Uploading: '1-6.PNG' which is 4452 bytes.
Uploading: '1-7.PNG' which is 4854 bytes.
Uploading: '1-8.PNG' which is 5555 bytes.
Uploading: '1-9.PNG' which is 5465 bytes.
```

图16-9

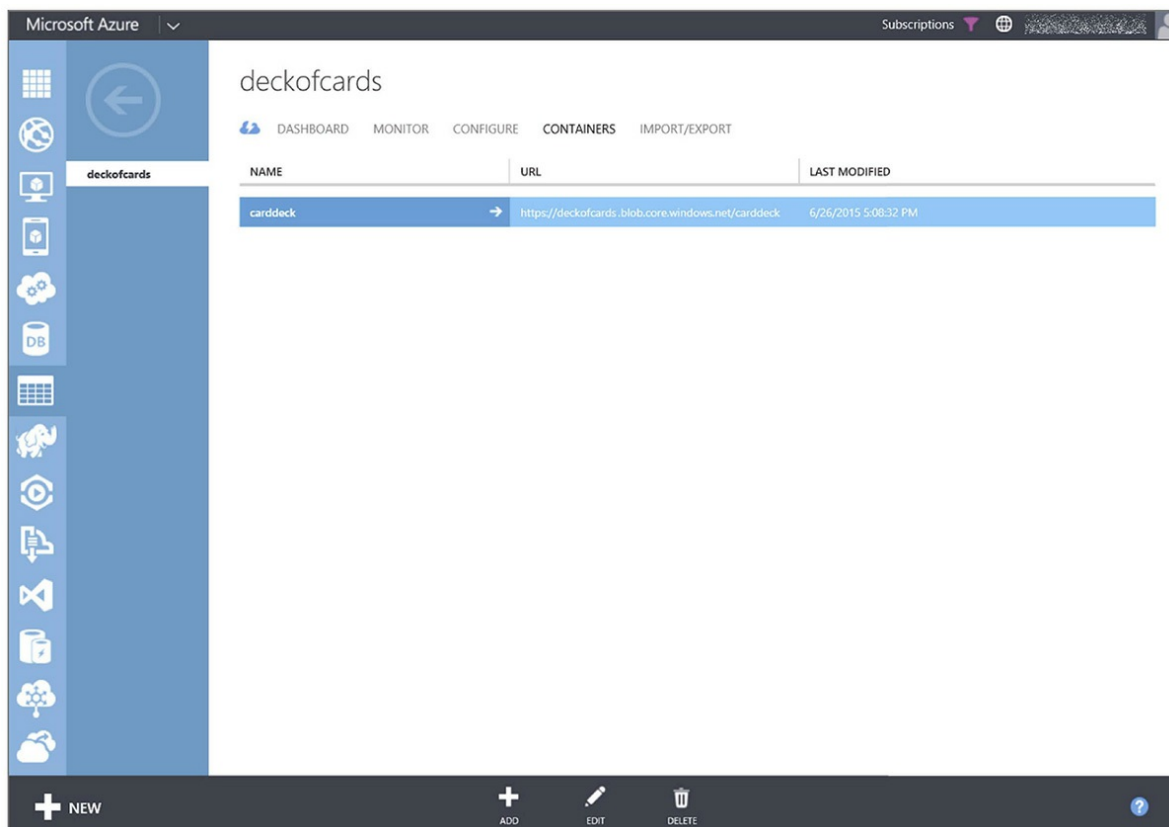


图16-10

示例说明

以编程方式可以创建Microsoft Azure存储账户，但这种创建方式的安全方面相对复杂，该步骤是直接在Microsoft Azure管理控制台上执行的。创建一个Azure存储账户后，就可以在该账户内创建多个容器。在本例中，创建了一个名为carddeck的容器。每个Microsoft Azure订阅的存储账户数量是有限的，而存储账户内的容器数量没有限制。可以创建任意多个容器，但要注意，每个容器都是有成本的。

本例的代码分为4部分（创建容器，把图片上传到容器，列出容器中的blob，可以根据需要选择删除容器的内容）。执行的第一步是为控制台应用程序建立try{ }...catch{ }框架。这是一种很好的实践方式，因为未捕获或未处理的异常通常会使过程（EXE）崩溃，而这总是应该避免的。第一个catch()表达式是StorageException，捕获在Microsoft.WindowsAzure.Storage名称空间的方法中专门抛出的异常。

```
catch (StorageException ex)
```

然后有一个捕获所有异常的表达式，处理所有其他意想不到的异常，并把异常消息写到控制台。

```
catch (Exception ex)
```

try{}代码块内的第一行代码使用添加到App.config文件中的细节来创建存储账户。

```
CloudStorageAccount storageAccount =  
    CloudStorageAccount.Parse(CloudConfigurationManager.GetSetting  
        ("StorageConnectionString"));
```

App.config文件包含在Azure存储账户上执行管理操作所需的存储账户名称和存储账户密钥。接下来，创建一个客户端程序，管理存储账户内特定blob容器的接口。然后代码获得特定容器carddeck的一个引用。

```
CloudBlobClient blobClient = storageAccount.CreateCloudBlobCli
CloudBlobContainer container =
blobClient.GetContainerReference("carddeck");
```

接下来调用container.CreateIfNotExists()方法。如果创建了容器，调用该方法意味着它不存在，返回true值，并把信息写入控制台。否则如果容器已经存在，就返回false。

```
if (container.CreateIfNotExists())
{...} ...
```

容器可以是Private或Public。对于这个示例，容器是Public，这意味着不需要访问键，就可以访问它。执行下面的代码，就可以把容器设置为Public:

```
container.SetPermissions(new BlobContainerPermissions
    { PublicAccess = BlobContainerPublicAccessT
```

现在创建了容器，并可以公开访问，但它是空的。使用像DirectoryInfo和FileInfo这样的System.IO方法，可以创建一个foreach循环，把每张扑克牌图片添加到carddeck存储容器中。GetBlockBlobReference()方法用于为要添加到容器中的特定图片名称设置引用。然后System.IO.File.OpenRead()方法利用文件名和路径，将实际文件打开为FileStream，并通过UploadFromStream()方法上载到容器中。

```

CloudBlockBlob blockBlob = container.GetBlockBlobReference(f.Name);
using (var fileStream = System.IO.File.OpenRead(@"Cards\" + f.Name))
{
    blockBlob.UploadFromStream(fileStream);
}

```

迭代Cards目录中的所有文件，并上传至容器。使用在carddeck容器的初始创建过程中创建的同一个容器对象，通过调用ListBlob()方法，把现有的一组blob返回为IEnumerable<IListBlobItems>。然后遍历列表，把它们写到控制台。

```

foreach (IListBlobItem item in container.ListBlobs(null, false))
{
    if (item.GetType() == typeof(CloudBlockBlob))
    {
        CloudBlockBlob blob = (CloudBlockBlob)item;
        WriteLine($"Card image url '{blob.Uri}' with length of " +
            $" {blob.Properties.Length}");
    }
    numberOfCards++;
}

```

如前所述，许多类型的项都可以存储在容器中，例如blob、表、队列和文件。因此在把项装箱到CloudBlockBlob之前，一定要确认该项是CloudBlockBlob。其他要检查的类型是CloudPageBlob和CloudBlobDirectory。

要删除容器中的blob，当遍历它们并写到控制台时，blob列表的检

索方式要与以前的相同。删除它们时的区别是调用 `GetBlockBlobReference (blob.Name)` 以获得特定blob的引用，然后给该blob调用 `Delete()` 方法。

```
CloudBlockBlob blockBlobToDelete = container.GetBlockBlobRefer  
blockBlobToDelete.Delete();
```

现在创建了Microsoft Azure存储账户和容器，加载了52张扑克牌的图像，所以可以创建一个ASP.NET网站来引用Microsoft Azure存储容器。

16.4 创建使用存储容器的ASP.NET 4.6网站

到目前为止，还没有深入探讨什么是Web应用程序，也没有讨论ASP.NET的基本方面。本节提供了一些洞察这些技术的视角。

Web应用程序让Web服务器向客户机发送HTML代码。这些代码显示在Web浏览器上，例如Internet Explorer。当用户在浏览器中输入URL字符串时，HTTP请求会被发送到Web服务器。HTTP请求包含所请求的文件名和其他信息，比如识别客户端应用程序的字符串、客户端支持的语言以及属于请求的其他数据。Web服务器返回一个包含HTML代码的HTTP响应，这些代码由Web浏览器解释，给用户显示文本框、按钮和列表。

ASP.NET是一个用服务器端代码动态创建Web页面的技术。这些Web页面的开发方式与客户端Windows程序有诸多相似之处。如果不直接处理HTTP请求和响应，手动创建发送到客户端的HTML代码，还可以使用创建HTML代码的控件，例如文本框、标签、组合框、创建日历的Calendar。

本书使用Web应用程序和ASP.NET，详细讨论这些主题超出了本书的范围。然而，本节将简要论述如何使用ASP.NET运行库，如何创建使用存储容器的ASP.NET网站。

为给客户端系统上的Web应用程序使用ASP.NET，只需要一个简单的Web浏览器。可使用Internet Explorer、Chrome、Firefox或其他任何支

持HTML的Web浏览器。客户端系统不需要安装.NET。

在服务器系统上，需要ASP.NET运行库。如果系统上有IIS，安装.NET Framework时就会用服务器配置ASP.NET运行库。在开发期间，不需要使用IIS，因为Visual Studio提供了自己的ASP.NET Web开发服务器，可以用它测试和调试应用程序。

为理解ASP.NET运行库是如何工作的，考虑一个来自浏览器的典型Web请求（参见图16-11）。客户端向服务器请求一个文件，如default.cshtml。ASP.NET Web窗体页面通常的文件扩展名是aspx（尽管ASP.NET MVC没有特定的文件扩展名），而cshtml用于基于Razor的网站。因为这些文件的扩展名用IIS注册，或者ASP.NET Web开发服务器能识别它们，所以ASP.NET运行库和ASP.NET工作进程就会启动。IIS工作进程命名为w3wp.exe，驻留在Web服务器的应用程序上。第一次请求default.cshtml时，启动ASP.NET解析器，编译器编译文件和C#代码，这些C#代码与.cshtml文件相关，并创建一个程序集。然后.NET运行库的JIT编译器把程序集编译为本机代码。之后销毁Page对象。但程序集保留下来，用于后续请求，所以没必要再次编译程序集。

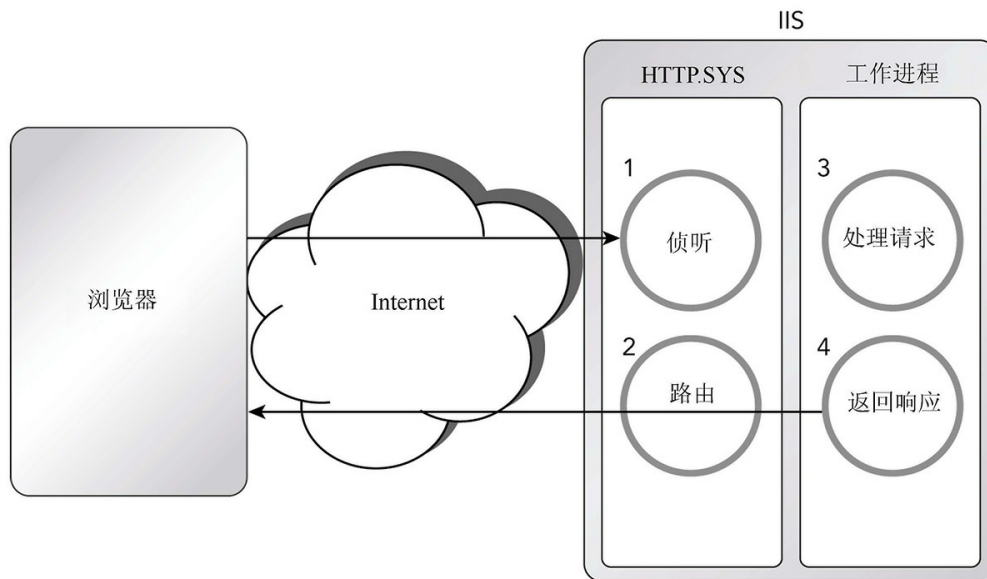


图16-11

基本理解Web应用程序和ASP.NET后，就可以执行下面示例中的步骤了。

试一试：创建一个**ASP.NET 4.6**网站来处理两手扑克牌

下面再次使用Visual Studio 2015，但这次要创建一个ASP.NET网站，请求两个玩家的名字，然后在提交页面时，处理两手扑克牌。这些扑克牌从之前创建的Microsoft Azure存储容器下载，扑克牌显示在Web页面上。

(1) 在Visual Studio中选择File|New|Web Site...，创建一个新的Web Site项目。在New Web Site对话框（参见图16-12）中，选择Visual C#类别，然后选择ASP.NET Empty Web Site模板。将网站命名为

Ch16Ex02。

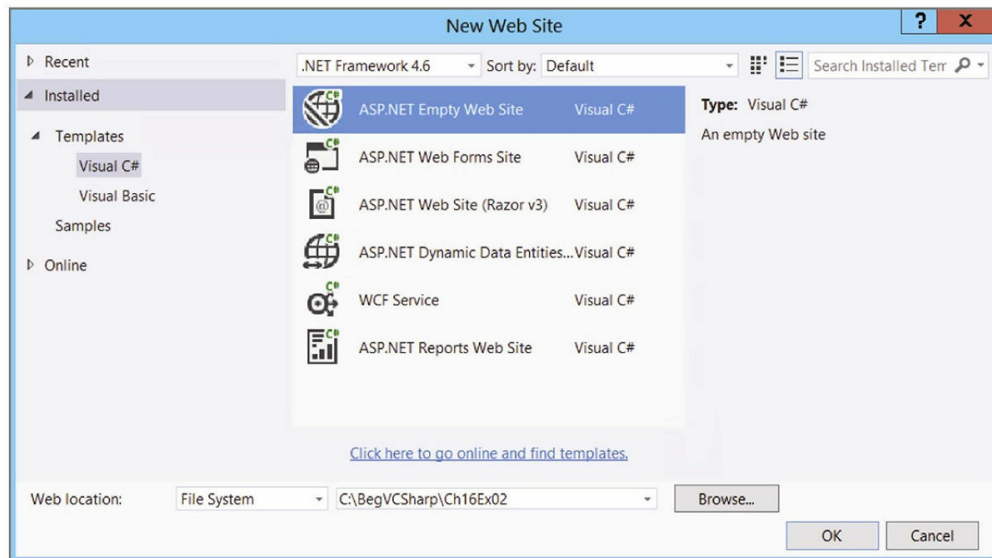


图16-12

(2) 右击Ch16Ex02解决方案，然后选择Add|Add ASP.NET Folder|App_Code，添加一个ASP.NET文件夹App_Code。

(3) 从下载站点下载示例代码，并将下面的类文件放在刚才创建的文件夹App_Code中。下载完毕后，右击App_Code文件夹，选择Add|Existing Item...，并从下载的示例中选择7个类。

- a. Card.cs
- b. Cards.cs
- c. Deck.cs
- d. Game.cs
- e. Player.cs

f. Rank.cs

g. Suit.cs

注意： 步骤（3）中的类非常类似于以前例子使用的类。只进行了少量调整，如移除WriteLine()、ReadLine()方法和一些未使用的方法。在Card.cs中有一个新的构造函数，其中包含到扑克牌图片的链接。

（4）右击CH16Ex02解决方案，然后选择Add New Item...|Visual C#|Empty Page（Razor v3），如图16-13所示，给项目添加一个Razor v3文件default.cshtml。

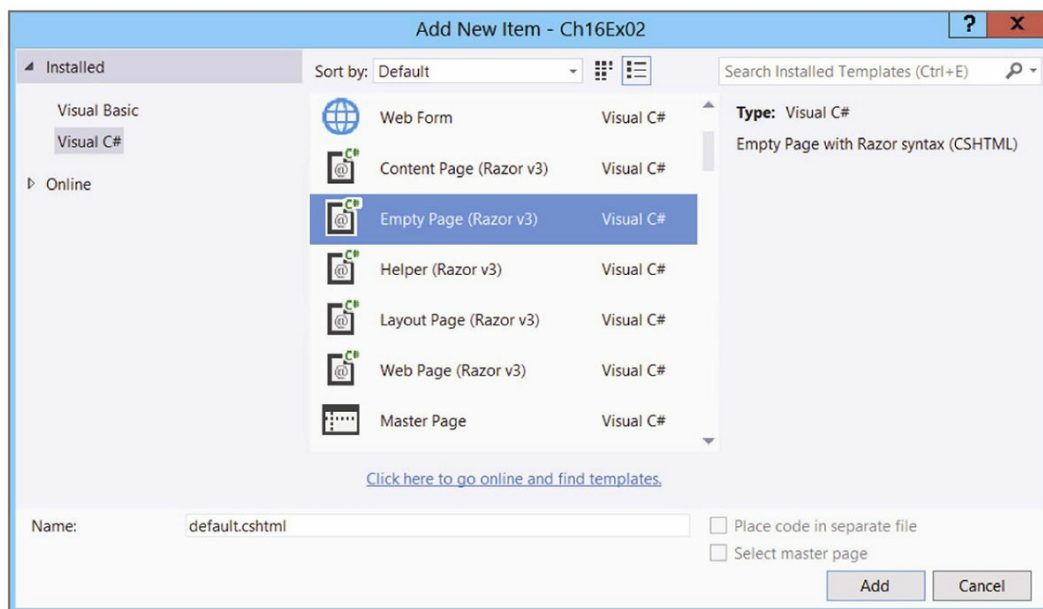


图16-13

(5) 打开default.cshtml文件，将下面的代码放在页面的顶部：

```
@{
    Player[] players = new Player[2];
    var player1 = Request["PlayerName1"];
    var player2 = Request["PlayerName2"];
    if(IsPost)
    {
        players[0] = new Player(player1);
        players[1] = new Player(player2);
        Game newGame = new Game();
        newGame.SetPlayers(players);
        newGame.DealHands();
    }
}
```

(6) 接下来，在第(5)步添加的代码之后添加如下语法。密切关注@card.image-link，这是给Card类新添加的参数。

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:40%; margin:1.2em;padding:10px;}
</style>
<title>BensCards: a new and exciting card game.</title>
```

```
</head>
<body>
    @if(IsPost){
        <label id="labelGoal">Which player has the best hand.</label>
        <br />
        <div>
            <p><label id="labelPlayer1">Player1: @player1</label></p>
            @foreach(Card card in players[0].PlayHand)
            {
                <img width="75px" height="100px" alt="cardImage"
                src=
                "https://deckofcards.blob.core.windows.net/carddeck/@card.i
                }
            </div>
            <div>
                <p><label id="labelPlayer1">Player2: @player2</label></p>
                @foreach(Card card in players[1].PlayHand)
                {
                    <img width="75px" height="100px" alt="cardImage"
                    src=
                    "https://deckofcards.blob.core.windows.net/carddeck/@card
                    }
                </div>
            }
        else
        {
            <label id="labelGoal">
```

```
        Enter the players name and deal the cards.  
    </label>  
<br /><br />  
    <form method="post">  
        <div>  
            <p>Player 1: @Html.TextBox(„PlayerName1")</p>  
            <p>Player 2: @Html.TextBox(„PlayerName2")</p>  
            <p><input type="submit" value="Deal Cards" class="submit  
        </div>  
    </form>  
    }  
</body>  
</html>
```

（7）现在，按F5键或Visual Studio中的Run按钮，运行Web站点。浏览器会启动，显示如图16-14所示的页面。首先提示输入玩家的名称。输入任意两个名字。

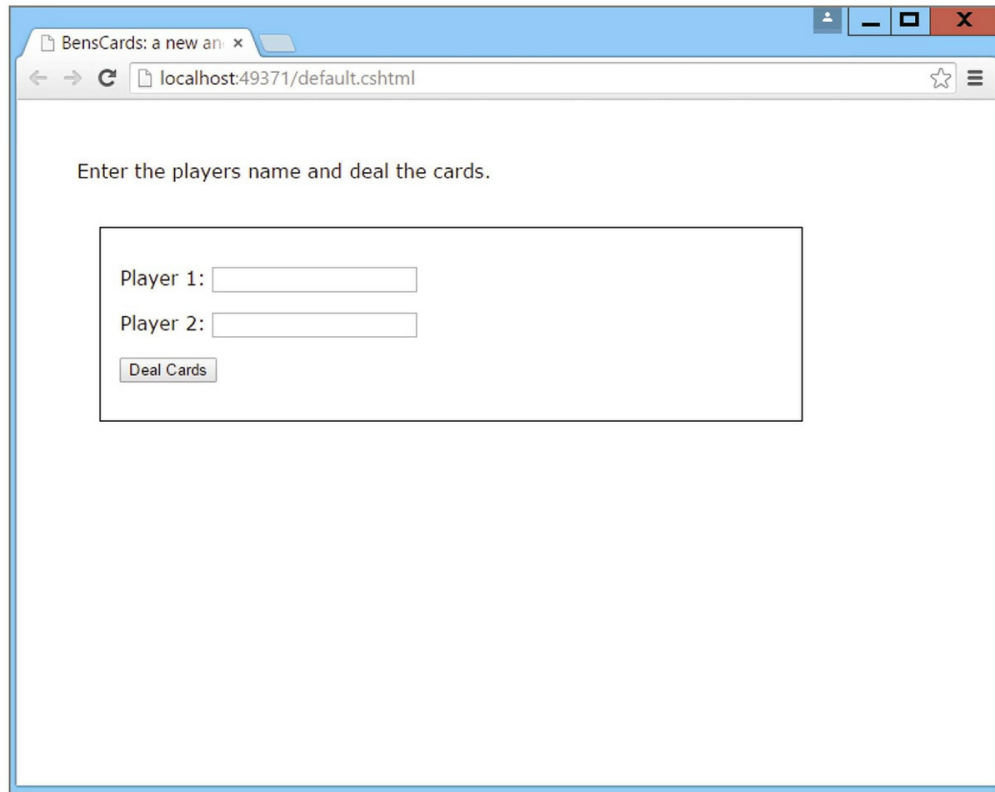


图16-14

(8) 按Deal Cards按钮，给每个玩家发一手牌。结果如图16-15所示。

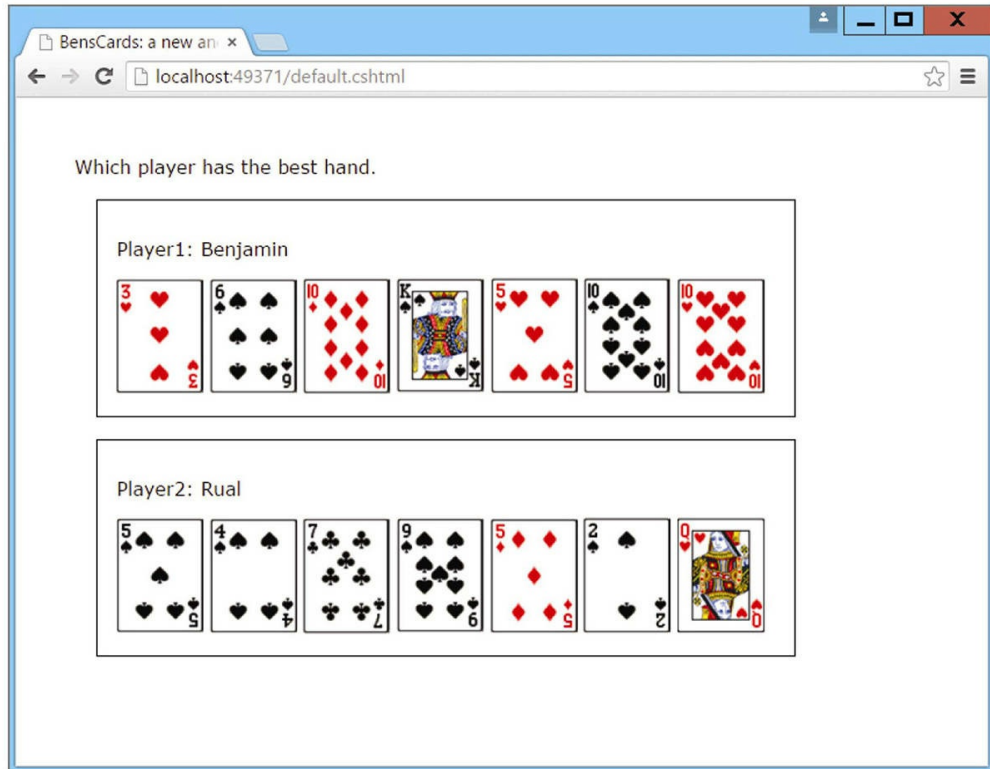


图16-15

前面使用Razor v3创建了一个简单的ASP.NET Web Site。ASP.NET Web Site连接到Azure存储账户和容器，以显示扑克牌的图像。

示例说明

前面的例子使用了一种名为Razor的新技术。Razor是一个与Visual Studio 2013与ASP.NET 3 MVC一起引入的视图引擎。Razor使用类似C#的语言（也支持VB），这些语言的代码放在@{...}代码块中，在浏览器请求页面时编译和执行。看看下面这段代码：

```
@{  
    Player[] players = new Player[2];  
    var player1 = Request["PlayerName1"];
```

```

var player2 = Request["PlayerName2"];
if(IsPost)
{
    players[0] = new Player(player1);
    players[1] = new Player(player2);
    Game newGame = new Game();
    newGame.SetPlayers(players);
    newGame.DealHands();
}
}

```

代码封装在一个@{...}代码块中，在访问时由Razor引擎编译和执行。访问该页面时，创建Player[]类型的数组，并把查询字符串的内容填充到两个变量player1和player2中。如果页面没有回送，就意味着只请求页面（GET），而不是单击按钮（POST），于是不执行if（IsPost）{}代码块内的代码。如果对页面的请求是一个POST，在单击Deal Cards时就会执行POST，实例化Players，开始一个新游戏，给玩家发一手牌。

第一次请求default.cshtml文件时，会执行如下代码路径，因为它不是一个POST。

```

else
{
    <label id="labelGoal">
        Enter the players name and deal the cards.
    </label>
    <br /><br />

```



```

    <form method="post">
      <div>
        <p>Player 1: @Html.TextBox("PlayerName1")</p>
        <p>Player 2: @Html.TextBox("PlayerName2")</p>
        <p><input type="submit"
              value="Deal Cards"
              class="submit">
        </p>
      </div>
    </form>
  }

```

下面的代码显示两个HTML文本框控件（用于请求玩家的名字）和一个按钮。一旦输入信息，就按下Deal Cards按钮，执行POST和随后的代码路径。代码遍历每个游戏玩家的牌。

```

@if (IsPost)
{
  <label id="labelGoal">Which player has the best hand.</label>
  <br />
  <div>
    <p><label id="labelPlayer1">Player1: @player1</label>
    @foreach (Card card in players[0].PlayHand)
    {
      <img width="75"
            height="100"
            alt="cardImage"

```

```

src=
"https://deckofcards.blob.core.windows.net/carddeck/@card.image
    }
</div>
<div>
    <p><label id="labelPlayer1">Player2: @player2</label>
    @foreach (Card card in players[1].PlayHand)
    {
        <img width="75"
            height="100"
            alt="cardImage"
src=
"https://deckofcards.blob.core.windows.net/carddeck/@card.image
        }
    </div>
}

```

注意，在两个foreach循环里，引用了Azure存储账户URL和前面练习中创建的容器。

注意： Azure存储账户URL和容器只是例子。应该用自己的Azure存储账户替换deckofcards，用自己的Azure存储容器替换carddeck。

16.5 练习

(1) 玩纸牌游戏时，需要在浏览器和服务端之间传送什么信息吗？

(2) 因为Web应用程序是无状态的，请说出存储这些信息的一些方法，使它们可包含在Web请求中。

16.6 本章要点

主题	要点
定义云	云是一个商品化的计算机硬件的弹性结构，用于运行程序。这些程序在混合云、公共云或私人云的类型中，在IaaS、PaaS或SaaS服务模型上运行
定义云优化堆栈	云优化堆栈是一个概念，指代码吞吐量高，占用空间小，可以与其他应用程序一起运行在同一台服务器上，并支持跨平台
创建存储账户	存储账户可包含数量不限的容器。存储账户是一种机制，用于控制访问其中创建的容器
用C#创建存储容器	存储容器存在于存储账户中，包含blob、文件或在互联网连接的情况下可从任何地方访问的数据
在ASP.NET Razor中引用存储容器	可在C#代码中引用存储容器。使用存储账户名、容器名、博客名、文件或需要访问的数据

第17章 高级云编程和部署

本章内容：

- 创建ASP.NET Web API
- 在Microsoft Azure上部署和使用ASP.NET Web API
- 在Microsoft Azure上缩放ASP.NET Web API

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 17 Code后，可找到与本章示例对应的单独文件。

前面已经花了一些时间学习云和云编程，下面再进一步，编写一些比上一章更复杂的C#代码。本章将继续探索ASP.NET和Microsoft Azure，修改CardLib程序，作为一个ASP.NET Web API运行在云上。一旦部署，就可在ASP.NET Web Site上使用它。

注意： 要成功完成本章的练习，需要一个Microsoft Azure订阅。如果没有一个，可以在<http://azure.microsoft.com>上注册一个免费试用

版。这十分便捷。

在创建、部署和使用ASP.NET Web API后，将学习如何缩放它。缩放概念很重要，掌握它会使自己创建的云程序更受欢迎。本章中的示例使用免费的Microsoft Azure云资源。这些免费资源有较低的CPU、内存和带宽阈值，在高使用率下很容易突破。本章将学习如何适时地缩放云程序，从而避免由于突破资源阈值造成云程序暂停。

17.1 创建ASP.NET Web API

计算机编程概念“应用程序编程接口（API）”已经存在几十年，通常描述为一个模块，它包含一组可用于构建软件程序的函数。

最初，从Windows客户机应用程序的角度看，这些模块是动态链接库（.dll），可以以编程方式访问的接口向其他程序公开内部的函数。在这样的系统中，当消费程序使用API时，它就会依赖接口的模式。修改接口，会导致消费程序异常和失败，因为访问和执行模块内函数的当前过程不再有效。一旦程序依赖一个接口，它就不应该改变，当它改变时，该事件就通常称为DLL Hell。有关DLL Hell的更多信息，可以阅读文章<http://www.desaware.com/tech/dllhell.aspx>。

随着时间的推移，互联网和内联网解决方案的实现成为主流，也实现了一些依赖技术，如Web服务和Windows Communication Foundation（WCF）。Web服务和WCF呈现了正式协定的接口，向其他程序公开包含在其中的函数。在前面提到的DLL API中，模块和使用它的程序在同一台计算机上，而Web服务和WCF在一个Web服务器上托管。由于托管在一个互联网或局域网Web服务器上，因此访问Web接口不再局限于一台计算机，而可以是任何设备，从任何有互联网或内联网连接的地方访问。

回顾前一章分析的云优化堆栈。在讨论中提到，为进行云优化，程序必须占用空间小，能够处理高吞吐量，支持跨平台。ASP.NET Web API基于ASP.NET MVC（模型、视图、控制器）的概念，这与新云优化堆栈的定义一致。如果已经创建了Web服务或WCF，或者过去使用过，

就将看到，ASP.NET Web API相对而言更简单、紧凑。如果从未使用它们，也能体会到这一点。

接下来的示例将创建一个ASP.NET Web API，处理一手牌。

试一试：创建ASP.NET Web API

下面使用Visual Studio 2015创建一个ASP.NET Web API，它接受一个玩家的名字，给该玩家返回一手牌。

（1）在Visual Studio中选择File|New|Project...，创建一个新的ASP.NET Web API。在New Project对话框中（参见图17-1），选择类别Visual C#|Web，然后选择ASP.NET Web Application模板。更改路径为C:\BegVCSharp\Chapter17\，把Web应用程序命名为Ch17Ex01，然后单击OK按钮。

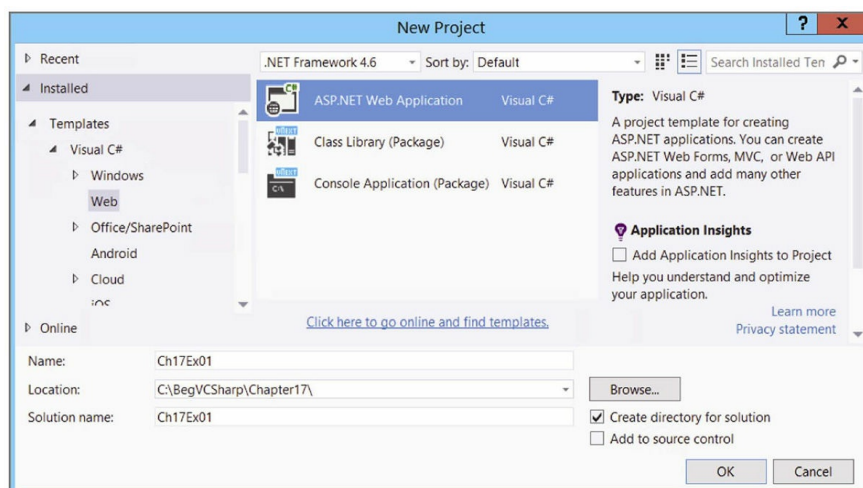


图17-1

(2) 接下来单击Empty ASP.NET 4.6 Template，选中Web API复选框，将需要的文件夹和核心引用添加到项目中。参见图17-2。现在取消选择Host in the cloud复选框（Web API的发布将在本章后面完成）。单击OK按钮。

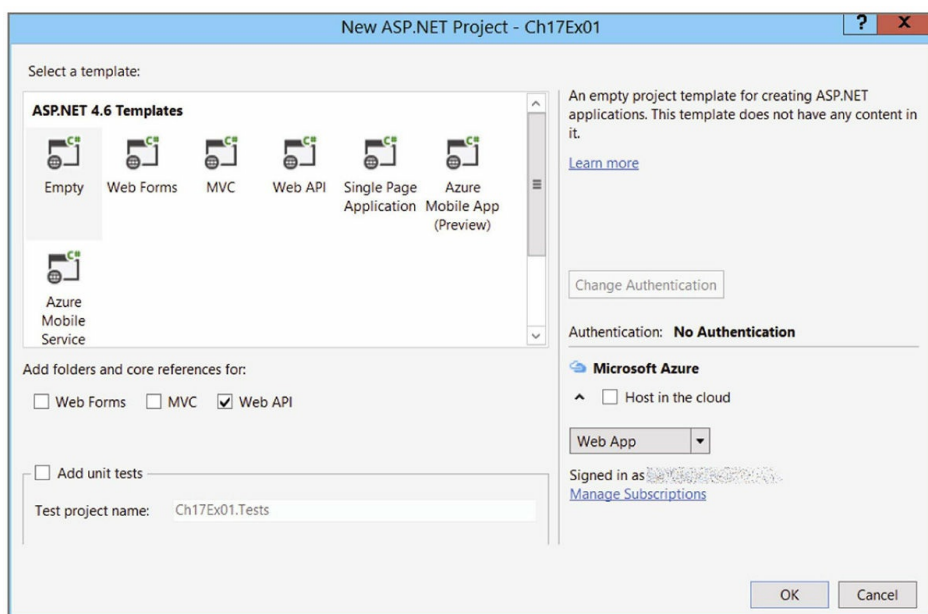


图17-2

(3) 右击Ch17Ex01解决方案，然后选择Add|New Folder|Rename，添加一个新文件夹CardLib。

(4) 从下载站点下载示例代码，将下面的类文件放在刚才创建的文件夹CardLib中。下载完成后，右击CardLib文件夹，并选择Add|Existing Item...，再从下载的示例中选择7个类。

a. Card.cs

b. Cards.cs

c. Deck.cs

d. Game.cs

e. Player.cs

f. Rank.cs

g. Suit.cs

注意： 这7个类与Ch16Ex02中使用的7个类一样。如果前面的练习已经下载了源代码，也可以在这里重用它们。

(5) 下面添加一个控制器：右击Controllers文件夹，选择Add|Controller...，再选择Web API 2 Controller- Empty...|Add（见图17-3）。

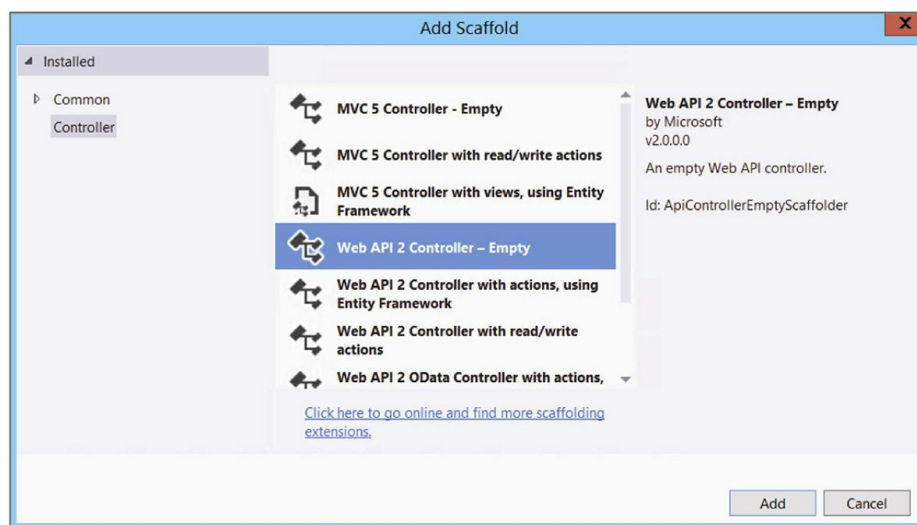


图17-3

(6) 把控制器命名为HandOfCardsController。

(7) 将如下代码添加到HandOfCardsController类中：

```
[Route("api/HandOfCards/{playerName}")]
public IEnumerable<Card> GetHandOfCards(string playerName)
{
    Player[] players = new Player[1];
    players[0] = new Player(playerName);
    Game newGame = new Game();
    newGame.SetPlayers(players);
    newGame.DealHands();
    var handOfCards = players[0].PlayHand;
    return handOfCards;
}
```

(8) ASP.NET Web API现在创建好了，准备发布到云。

恭喜！前面创建了返回一手牌的ASP.NET Web API。

示例的说明

创建新的ASP.NET Web API时，有两个选项。这个示例使用了第一种方法。从模板选择窗口选择Empty，表示这个项目只包含创建ASP.NET Web API所需的基本内容，因此添加到解决方案中的配置文件和二进制文件很少。这个Web API占用的空间非常小，这正是在云中优化运行所需要的。

另一种可能的方法是选择Web API模板（参见图17-2）而不是Empty。这包括额外的配置文件、许多额外的引用以及ASP.NET MVC应用程序的一个基本例子。这个示例相对较小，不需要任何MVC功能，所以选择了Empty模板。如果在未来自己的项目中需要额外的功能和例子，就考虑选择Web API模板，因为它构建了数据管道，提供了许多已证明有效的编码模式来构建解决方案。

这里把第16章中示例使用的7个类添加到CardLib目录中。GetHandOfCards()方法的内容与第16章相同。该方法接受一个参数playerName，创建一个新游戏，设置玩家，发牌，并给ASP.NET Web API消费程序返回Cards类。添加的一行代码如下：

```
[Route("api/HandOfCards/{playerName}")]
```

Route注解说明ASP.NET如何决定哪个Web API方法响应哪个请求。在发布后，与ASP.NET Web API交互时，并没有请求具体的文件。在ASP.NET Web Forms应用程序中，请求发送给扩展名为.aspx的文件，但在调用Web API（或ASP.NET MVC应用程序）时，就不是这样。Web API请求发送到Web服务器，其参数在请求的URL中，用正斜杠分隔。例如：

```
http://contoso.com/api/{controllerName}/Parameter1/Parameter2/
```

注意：不使用注解（annotation）来创建路由地图，而可在App_Start目录的WebApiConfig.cs中创建它们。

现在创建了ASP.NET Web API，下一节学习Web API的部署和使用。

17.2 在Microsoft Azure上部署和使用ASP.NET Web API

有很多选项可用于将Web应用程序部署到Microsoft Azure平台上。最流行的方法之一是使用本地Git存储库或托管在GitHub上的公共Git存储库。本地和公共Git存储库都提供了版本控制功能，这是一个非常有用的功能。版本控制允许开发人员和发布经理了解进行了什么改变，并了解进行这些修改的时间和操作人员。当编译二进制文件，或把更改部署到现场的环境中时，如果有问题或未预料到的异常，就很容易找到联系人。可以整合到Microsoft Azure上的其他部署平台包括Team Foundation Services、CodePlex和Bitbucket。

注意：可采用许多方法把代码部署到Microsoft Azure平台。存储在源代码存储库中的项目与具体的独立代码场景都有多个独立部署选项。

之前示例中的代码是一个独立项目，不包含在版本控制库中，在IDE中直接执行部署，在本例中是在Visual Studio 2015中部署。部署不包含在源代码存储库中的解决方案的其他方法包括Web部署（msdeploy.exe）和FTP。

完成以下示例，把ASP.NET Web API部署到Microsoft Azure Web App上。

试一试：把ASP.NET Web API部署到云上

(1) 右击Ch17Ex01项目|Publish...。

(2) 选择Microsoft Azure Web App|Manage订阅，导入自己的Microsoft Azure订阅（如果需要）。

(3) 选择New...按钮，创建一个新的Microsoft Azure Web App，输入需要的值，按下Create按钮（图17-4）。

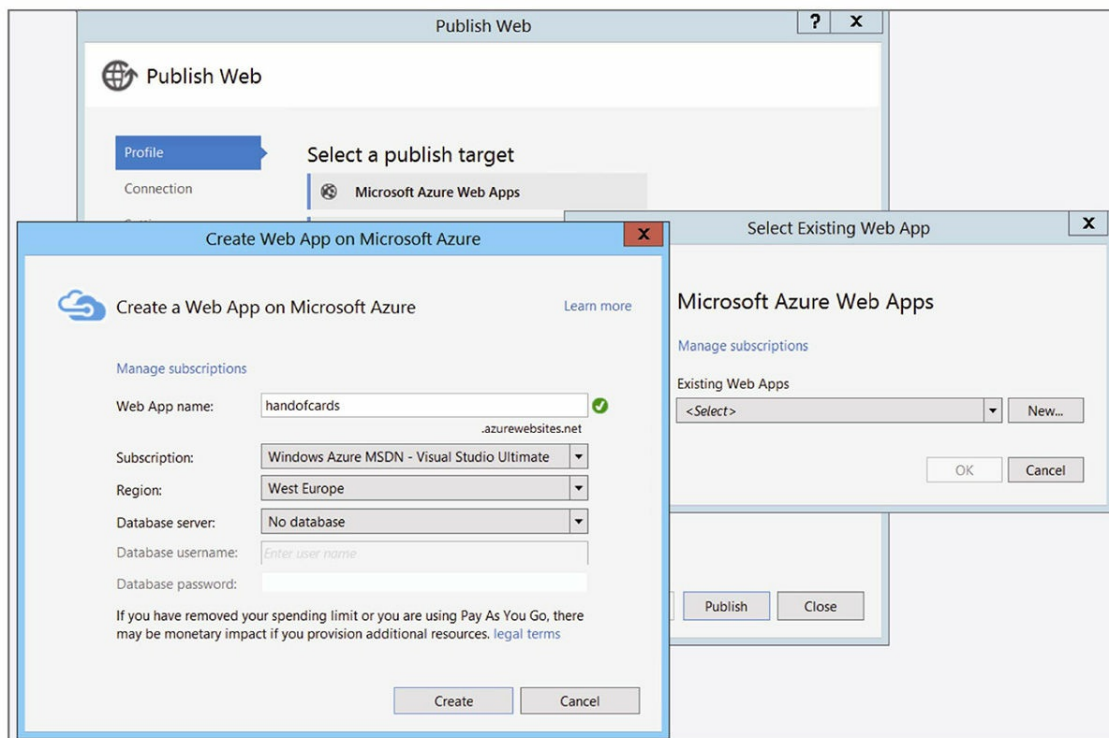


图17-4

- Web App Name: 必须是唯一的名称。

- **Subscription:** 如果有多个Microsoft Azure订阅，就选择希望创建这个Web App的订阅。
- **Region:** 选择要创建Web App的地点。
- **Database server:** 可在发布Web App的过程中创建数据库。在本例中，不需要数据库。

(4) 一旦创建完毕，就会显示Publish Web窗口（见图17-5）。按下Publish按钮，把ASP.NET Web API部署到云上。在发布之前，可考虑按下Validate Connection按钮，确保配置和凭据的正确建立。

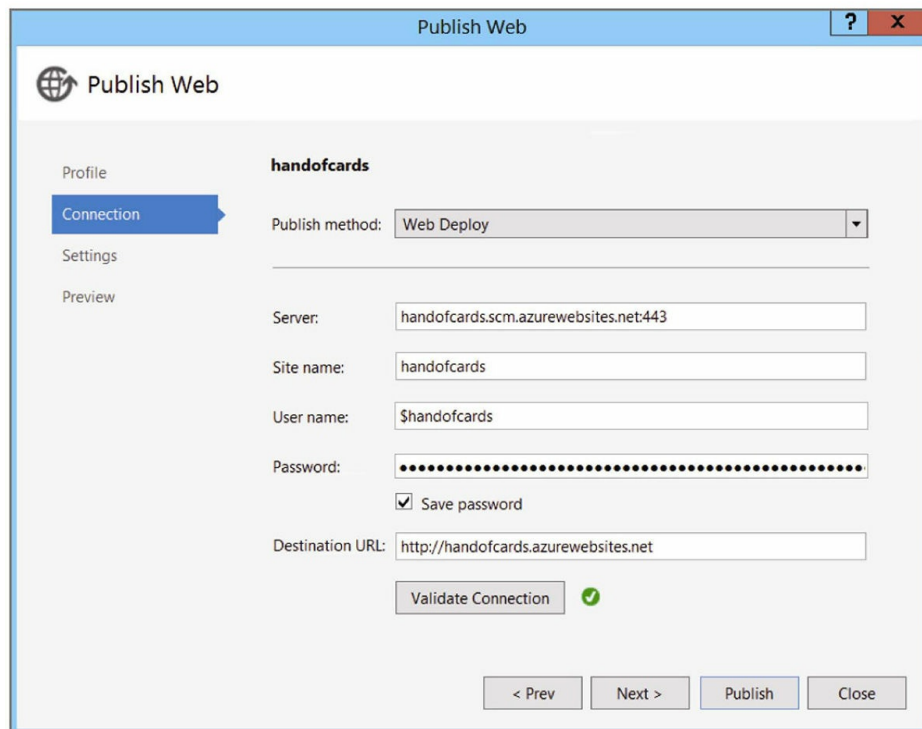


图17-5

(5) 在ASP NET Web API成功发布后，浏览器将打开，通知Web应用程序已成功创建。也可以在Visual Studio的Output窗口中查看详细信息，找到发布步骤的更多信息。

(6) 检查ASP.NET Web API的响应。例如，现在可通过<http://handofcards.azurewebsites.net/api/HandOfCards/Benjamin>进行全局访问，其中handofcards是创建Microsoft Azure Web App时提供的名称，Benjamin是玩家的名字。

注意：默认情况下，不同的浏览器以不同方式显示结果。例如，Internet Explorer提示下载一个JSON文件，而Chrome浏览器显示一些XML数据。重要的是我们得到了响应。API的用法参见下一节。

示例的说明

在Visual Studio中发布Web App时，它在后台使用Web Deploy执行实际的部署。知道了这一点，如果有特殊的部署要求，它们可以在Properties\PublishProfiles目录的发布配置文件中设置。*.pubxml的内容包含给定部署的配置项和依赖关系。

部署完成后，浏览器会显示Web App的主页（见图17-6），而不是ASP.NET Web API。

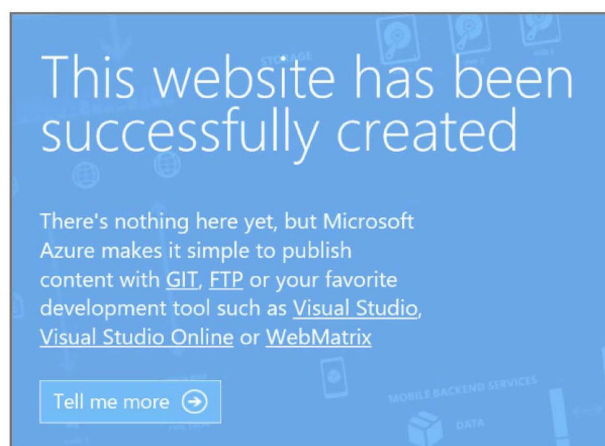


图17-6

与包含在.dll、Web服务或WCF服务中的传统API不同，直接访问ASP.NET Web API实际上并不常见。相反，在所有情况下，对API的调用来自包含在另一个（使用API的）项目或解决方案的代码中。

现在部署了ASP.NET Web API，此后，就可以在能发出HTTP请求、能解析JSON文件的任意客户端上使用它。以下示例提供的指令可用于学习如何在ASP.NET Web页面上使用刚才发布的ASP.NET Web API。

注意： 以下示例修改了Ch16Ex02 ASP.NET网站。主要区别是：它未从网站本身包含的类中检索Cards，而从本章创建和部署的ASP.NET Web API中检索。

试一试：在网站中使用**Web API**

下面使用Visual Studio 2015修改CH16Ex02例子，以使用ASP.NET Web API。Web API接受一个玩家的名字，并给玩家返回一手牌。

（1）打开Visual Studio 2015，选择File|New|Web Site，并从已安装模板的Visual C#列表中选择ASP.NET Empty Web Site。

（2）把Web Location改为c:\BegVCSharp\Chapter17\Ch17Ex02，然

后按OK按钮继续。

注意： ASP.NET Web API的输出是一个JSON文件，它的格式遵循一个使其容易解析的标准格式。解析JSON文件最常见的方式是使用Newtonsoft.Json库。

（3）为安装用于解析JSON文件的Newtonsoft.Json库，右击Solution，然后选择Manage NuGet Packages...，打开Visual Studio中的一个选项卡，如图17-7所示。

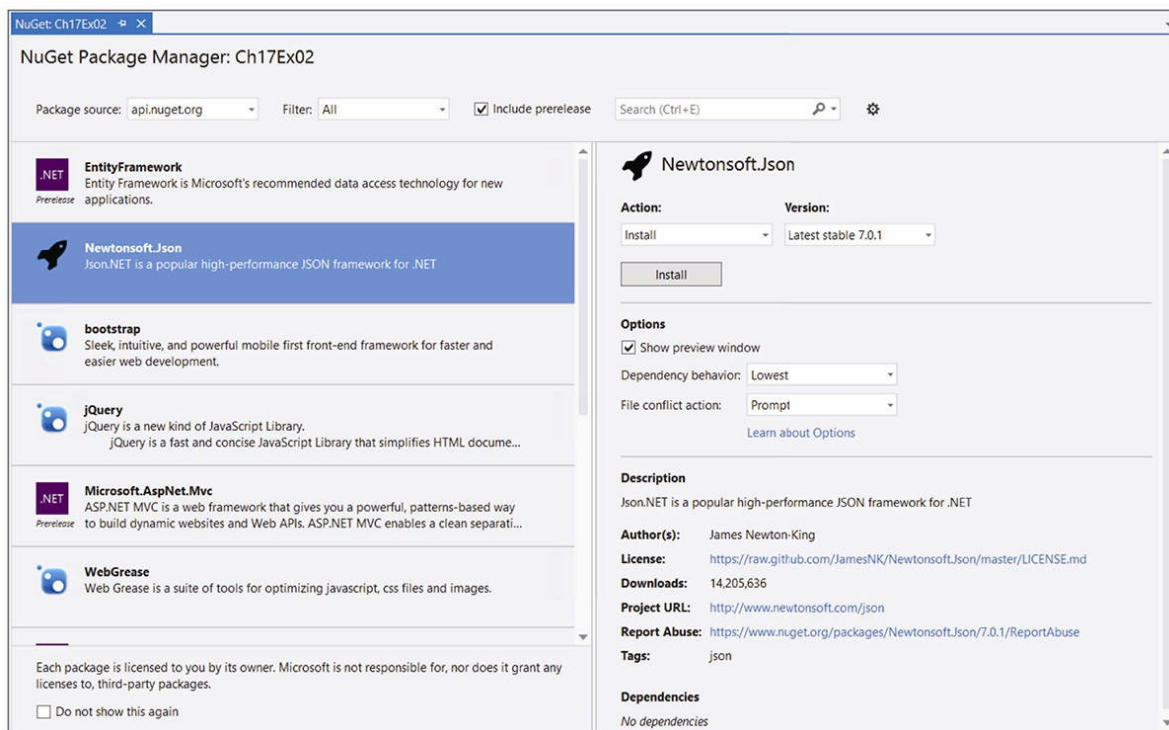


图17-7

（4）从Package列表中选择Newtonsoft.Json，并按Install按钮。把

Bin目录添加到ASP.NET Web Site中，以包含Newtonsoft.Json.dll库。

(5) 为把cshtml文件添加到解决方案中，右击Ch17Ex02，并选择Add|Add New Item...|Empty Page (Razor v3)，命名为default.cshtml，单击Add按钮。

注意： 这里，default.cshtml文件的内容与之前在CH16Ex02中创建的内容非常相似，但需要一些调整。考虑复制第16章中default.cshtml的内容，而不是重新输入整个页面。

(6) 接下来，在页面的顶端添加如下语句，把Newtonsoft.Json库包含到Razor文件中：

```
@using Newtonsoft.Json;
```

(7) 在第6步添加的代码后面添加如下代码段：

```
@{  
    List<string> cards = new List<string>();  
    var playerName = Request["PlayerName"];  
    if (IsPost)  
    {  
        string GetURL = "http://handofcards.azurewebsites.net/api/  
            "HandOfCards/" + playerName;  
        WebClient client = new WebClient();  
        Stream dataStream = client.OpenRead(GetURL);  
    }  
}
```

```

StreamReader reader = new StreamReader(dataStream);
var results =
    JsonConvert.DeserializeObject<dynamic>(reader.ReadLine
reader.Close());
foreach (var item in results)
{
    cards.Add((string)item.imageLink);
}
}
}

```

(8) 最后，在第7步添加的代码后面添加如下标记和Razor代码，以使用ASP.NET Web API:

```

<html>
<head>
    <title>BensCards: Deal yourself a hand.</title>
</head>
<body>
    @if (IsPost)
    {
        <label id="labelGoal">Here is your hand of cards.</label>
        <br />
        <div>
            <p><label id="labelPlayer1">Player1: @playerName</labe
            @foreach (string card in cards)
            {

```

```

        <img width="75"
            height="100"
            alt="cardImage"
            src=
                "https://deckofcards.blob.core.windows.net/carddeck/((
        }
    </div>
    <label id="errorMessageLabel" />
}
else
{
    <label id="labelGoal">
        Enter the players name and deal the cards.
    </label>
    <br /><br />
    <form method="post">
        <div>
            <p>Player 1: @Html.TextBox("PlayerName")</p>
            <p><input type="submit" value="Deal Hand" class="su
        </div>
    </form>
}
</body>
</html>

```

(9) 按F5运行ASP.NET Web Site。界面呈现出来后，输入一个名称，按下Deal Hand按钮。ASP.NET Web Site就会使用ASP.NET Web

API，显示一手牌，如图17-8所示。

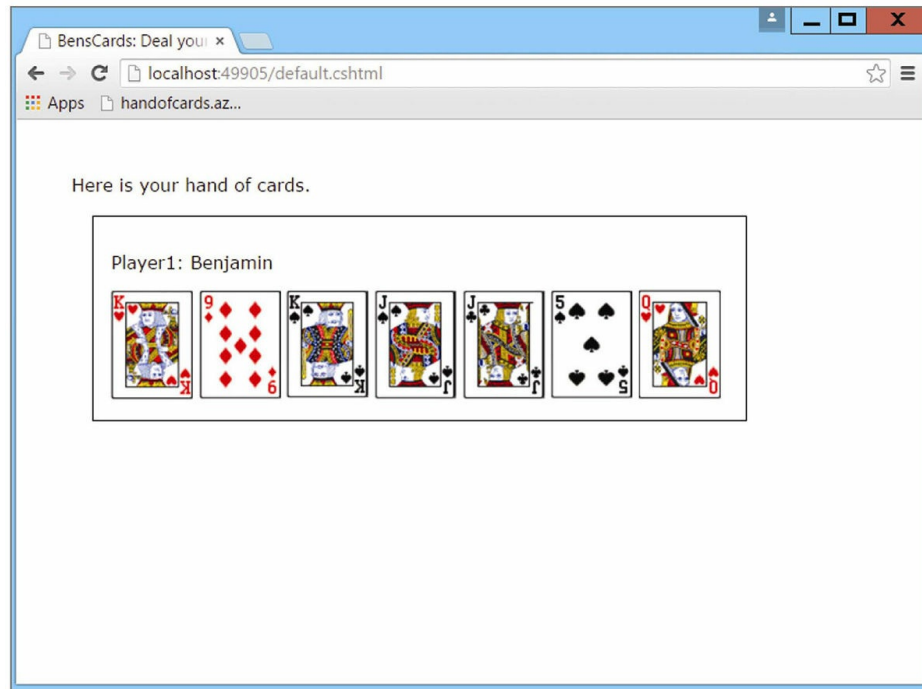


图17-8

示例的说明

最初显示default.cshtml页面时，IsPost是false，因此不执行在Razor代码块的C#代码中对ASP.NET Web API的调用。而只显示else代码块中的部分HTML代码。呈现部分包含一个捕获玩家名字的文本框，和一个触发把页面回送给自己的按钮。

一旦输入了玩家的名字，按下Deal Hand按钮，IsPost属性就变成true，执行页面顶部Razor标签中的C#代码。

```
string GetURL = "http://handofcards.azurewebsites.net/api/Hand  
    playerName;  
WebClient client = new WebClient();
```

```
Stream dataStream = client.OpenRead(GetURL);
```

GetURL字符串中存储的Web地址是ASP.NET Web API的互联网或内联网位置，用作OpenRead类中OpenRead()方法的一个参数。WebClient包含执行HTTP请求所需的方法。OpenRead()方法的结果存储在一个Stream对象中。

```
StreamReader reader = new StreamReader(dataStream);  
var results = JsonConvert.DeserializeObject<dynamic>(reader.Re
```

然后Stream对象作为一个参数传递给StreamReader构造函数。使用StreamReader类的ReadLine()方法作为参数，使用Newtonsoft.Json库来反序列化JSON玩家，结果可以通过foreach语句来枚举，添加到List<string>容器cards中。cards列表可以访问，用于后面的页面呈现过程。

```
foreach (var item in results)  
{  
    cards.Add((string)item.imageLink);  
}
```

注意： 回顾一下前面第13章讨论的dynamic类型。dynamic类型与JSON文件一起使用是很常见的，因为它包含的结构并不总是可以强制转换为强类型的类。

一旦JSON文件的解析结果加载到cards容器中，就执行IsPost代码块内的标记代码。Razor标签内的foreach循环读取cards容器，连接图

像名称和第16章创建的Microsoft Azure Blob Container链接。

```
@foreach (string card in cards)
{
    
}
```

可以考虑使用从之前示例中获得的知识把这个ASP.NET Web Site部署Microsoft Azure平台上。例如，只需要右击Ch17Ex02解决方案，选择Publish Web App，并按照发布向导的指示进行。创建一个Web应用程序“handofcards-consumer”，将可以从<http://handofcards-consumer.azurewebsites.net/>访问它。ASP.NET Web API和Microsoft Azure Blob Container都可以在世界上的任何地方通过互联网访问，把ASP.NET Web Site放在Azure上，会得到相同的结果（获得一手牌）。

随着时间的推移，如果使用程序或API较受欢迎，开始收到很多请求，在FREE模式下运行的Web App可能导致资源阈值被突破，使资源不可用。这并不是最理想的。下一节将了解如何扩展在Microsoft Azure平台上运行为Web App的ASP.NET Web API，以使用户和客户可以在需要时访问可响应的Web资源。

17.3 扩展Microsoft Azure平台上的ASP.NET Web API

以前，扩展以满足用户的需求是一个非常繁杂、费时、昂贵的活动。从历史上看，当公司想增加服务器容量，支持更多的流量时，就需要采购、装配物理硬件，把物理硬件配置到数据中心。然后，一旦硬件进入网络，就交给应用程序所有者，安装并配置操作系统、所需的组件和应用程序的源代码。执行此类任务所需的时间很长，而公司安装了大量的物理容量，供高峰时间使用；然而，在非高峰使用时期，额外的容量却未使用，只能闲置，这是一个非常昂贵、非最优的资源分配方法。

更好的方法是使用云平台，像Microsoft Azure提供了在需要资源时利用物理资源向上、向下、向外扩展的最优能力。当需要物理资源（如CPU、磁盘空间或内存）时，就向上或向外扩展，来满足要求，当对云服务的需求减少时，可以缩小物理资源，把经费用于其他项目和服务。

注意：为成功完成本章的练习，需要一个Microsoft Azure订阅。如果没有，可在<http://azure.microsoft.com>上注册一个免费试用版，这十分便捷。

本章的其余部分说明了如何根据CPU的需求在具体的时间段里扩展ASP.NET Web API。

试一试：根据CPU的需求扩展ASP.NET Web API

(1) 访问Microsoft Azure门户网站
<https://manage.windowsazure.com>。

(2) 选择本章前面创建的ASP.NET Web API，例如handofcards。如图17-9所示，注意Web App在FREE定价层。只有当Web App在STANDARD模式时，Auto Scaling才可用。

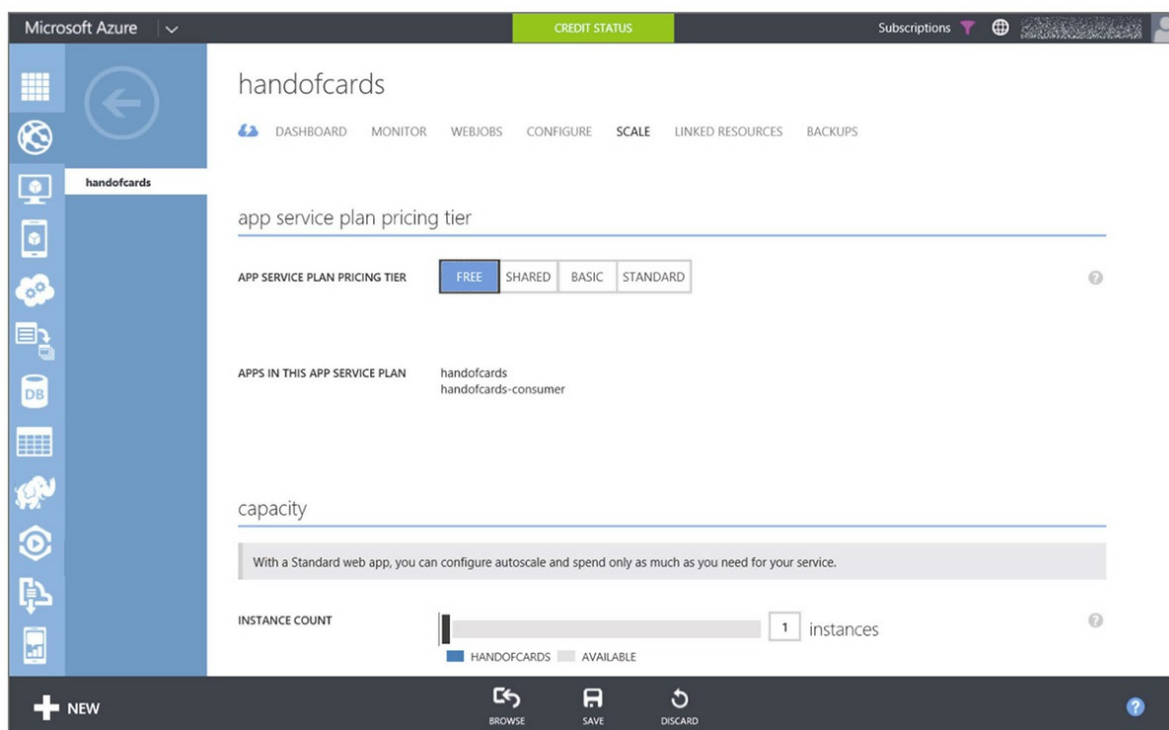


图17-9

(3) 单击STANDARD Tier框，再单击页面底部的Save按钮，把Web App向上扩展到STANDARD。

(4) 一旦保存了配置，就向下滚动，显示容量扩展选项（参见图17-10）。单击CPU SCALE BY METRIC设置。

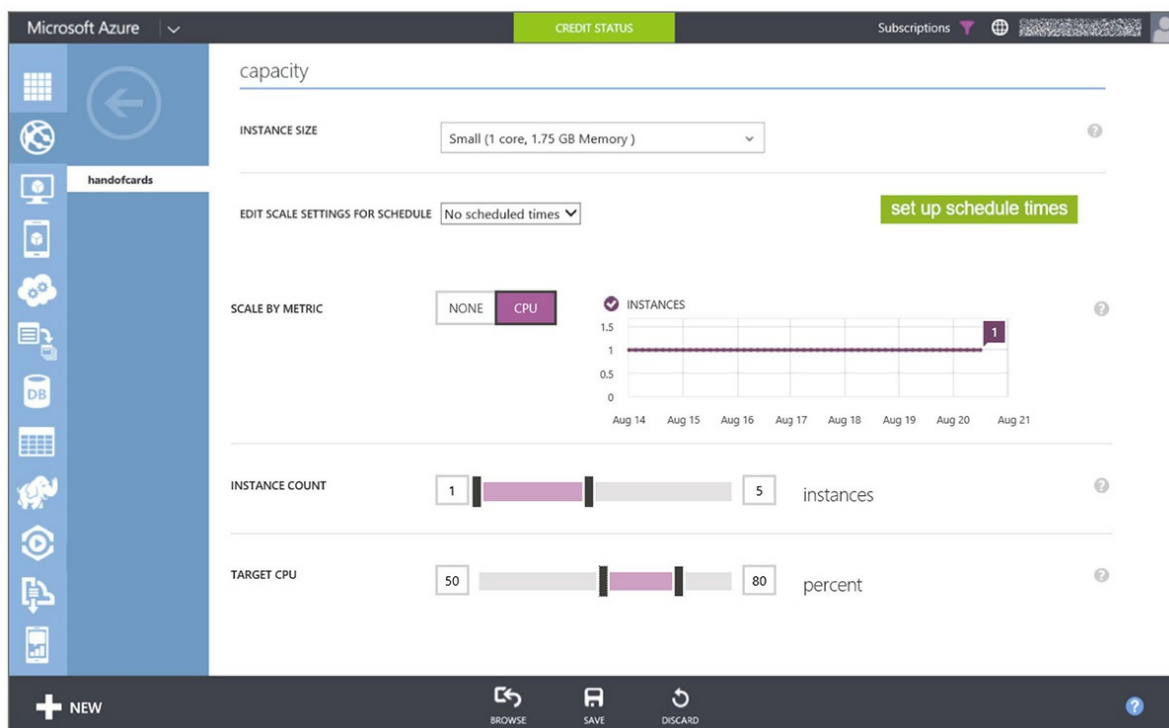


图17-10

(5) 把INSTANCE COUNT最大改为5，TARGET CPU改为50和80。

(6) 保存配置。

示例的说明

在FREE模式下运行Web App并不会得到很多。它其实用于测试和学习Microsoft Azure平台是如何工作的。Auto Scaling功能只在标准模式下可用，因此只有扩展到这一层，才能访问该功能。SHARED和BASIC

等其他模式有助于扩展的容量，但需要手动配置。

默认情况下，对于给定的60分钟时间，每5分钟检查一次，如果CPU利用率平均在60%和80%之间，Auto Scale设置最多可扩展到这个Web App的3个实例。

注意图17-10中INSTANCE SIZE下降了。默认情况下，实例所占空间很小，相当于1×2.6 GHz的CPU和1.75 GB的内存。这意味着，当扩展到三个实例时，会收到三个不同的虚拟机，每个虚拟机都占用了1×2.6 GHz的CPU和1.75GB的内存。如果INSTANCE SIZE设置为Large，就将得到三个虚拟机，每个虚拟机都占用了4×2.6 GHz的CPU和7GB的内存。

最后，当运行Web App的虚拟机上的CPU利用率突破了设置为TARGET CPU值的较低阈值时，就给环境添加一个新实例或虚拟机，直到INSTANCE COUNT设定的最大数量。

自动伸缩功能非常适于管理意想不到的Web App使用高峰期和对Web App的请求。然而，如果已知顾客或用户何时与Web App交互，就可以提前计划，在实际需要时，稍提前一点提供额外的实例。好处是不需要根据CPU的使用情况逐步增加或减少实例，而可以立即缩放到特定的时间段需要的CPU数量和内存量。例如，如果知道营销部门在10月搞一个活动，就可以在那个月安排额外的可用资源。使所需的资源可用并进行预热，就可以避免它们延迟分配给用户或客户使用。执行下面例子中的步骤，看看具体操作。

试一试：在特定的时间缩放ASP.NET Web API

(1) 访问Microsoft Azure门户网站
<https://manage.windowsazure.com>。

(2) 选择本章前面创建的ASP.NET Web API，例如handofcards。如图17-9所示，注意Web App在FREE定价层。只有当Web App在STANDARD模式下时，Auto Scaling才可用。

(3) 单击STANDARD Tier框，再单击页面底部的Save按钮，把Web App向上扩展到STANDARD。

(4) 一旦保存了配置，就向下滚动，显示容量扩展选项。单击set up schedule times链接，显示一个弹出窗口，如图17-11所示。

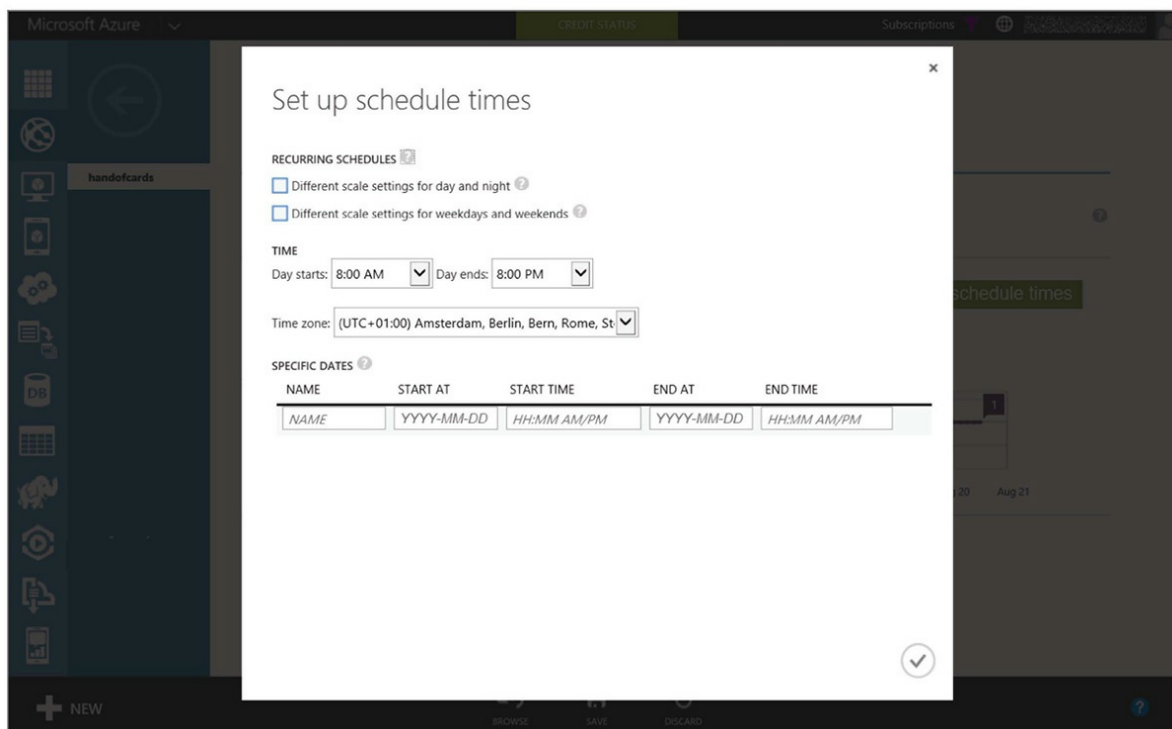


图17-11

(5) 输入名称、日期和时间，如图17-12所示。

SPECIFIC DATES ?				
NAME	START AT	START TIME	END AT	END TIME
October	2015-10-01	09:00 AM	2015-10-31	06:00 PM
<input type="text" value="NAME"/>	<input type="text" value="YYYY-MM-DD"/>	<input type="text" value="HH:MM AM/PM"/>	<input type="text" value="YYYY-MM-DD"/>	<input type="text" value="HH:MM AM/PM"/>

图17-12

(6) 在弹出窗口中单击复选标记，如图17-11所示。

(7) 在EDIT SCALE SETTINGS FOR SCHEDULE下拉表中选择预定缩放配置文件的名称。例如,选择October，如图17-13所示。然后，为选中的预定配置文件选择INSTANCES的数量，例如5。

The screenshot shows the Microsoft Azure portal interface. On the left is a navigation pane with icons for various services. The main area is titled 'capacity'. Below this, there's a section for 'INSTANCE SIZE' set to 'Small (1 core, 1.75 GB Memory)'. Below that is 'EDIT SCALE SETTINGS FOR SCHEDULE' with a dropdown menu showing 'October'. To the right of this dropdown is a green button labeled 'set up schedule times'. Below this is the 'SCALE BY METRIC' section with two buttons: 'NONE' and 'CPU', where 'CPU' is selected. To the right of these buttons is a line graph titled 'INSTANCES' showing a constant value of 1 from Aug 14 to Aug 21. Below the graph is the 'INSTANCE COUNT' section with a slider bar and a text input field showing '5 instances'.

图17-13

（8）单击页面底部的SAVE按钮，当配置的时间段到达时，缩放设置就会生效。

为扩展Web App而创建时间表时，需要名称、开始日期、开始时间、结束日期和结束时间。有了这些信息，Microsoft Azure平台就可以管理可用实例的数量，这些实例是在配置好的时间段内为Web App的请求提供服务的虚拟机。可以创建无数时间表，每个都有自己的实例数量和缩放设置。只要创建时间表，保存它，在需要时从schedule下拉框中选择它，资源就会按预期变成可用的。

17.4 练习

(1) 不是在ASP.NET Web Site应用程序中使用ASP.NET Web API，而是在另一个应用程序类型中使用它，例如控制台应用程序或Windows通用应用。

(2) 对于Microsoft Azure平台上的Web App，实例的最大大小和数量是多少？

17.5 本章要点

第IV部分 数据访问

- 第18章 文件
- 第19章 XML和JSON
- 第20章 LINQ
- 第21章 数据库

第18章 文件

本章内容：

- File和Directory类
- .NET如何使用流类访问文件
- 如何读写文件
- 如何读写压缩数据
- 如何序列化和反序列化对象
- 如何监控文件和目录的变化

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 18 Code后，可以找到与本章示例对应的单独文件。

文件是在应用程序的实例之间存储数据的一种便利方式，也可用于在应用程序之间传输数据。文件可以存储用户和应用程序配置，以便在下次运行应用程序时检索它们。

这一章展示如何在应用程序中有效地使用文件，涉及用于创建和读写文件的主要类，以及支持在C#代码中处理文件系统的类。本章不详细

介绍全部的类，但将深入介绍一些类，以便读者理解概念和基本理论。

18.1 用于输入和输出的类

读写文件是把数据送入C#程序（输入）和送出程序（输出）的基本方式。因为文件用于输入输出，所以文件类包含在System.IO名称空间中（IO是Input/Output的常见缩写形式）。

System.IO包含用于在文件中读写数据的类，只有在C#应用程序中引用此名称空间才能访问这些类，而不必完全限定类型名。

本章将介绍如表18-1所示的一些类。

表18-1 用于访问文件系统的类

类	说明
File	静态实用类，提供许多静态方法，用于移动、复制和删除文件
Directory	静态实用类，提供许多静态方法，用于移动、复制和删除目录
Path	实用类，用于处理路径名称
FileInfo	表示磁盘上的物理文件，该类包含处理此文件的方法。要完成对文件的读写工作，就必须创建Stream对象
DirectoryInfo	表示磁盘上的物理目录，该类包含处理此目录的方法
FileSystemInfo	用作FileInfo和DirectoryInfo的基类，可以使用多

	态性同时处理文件和目录
FileSystemWatcher	FileSystemWatcher是本章要介绍的最复杂类。它用于监控文件和目录，提供了这些文件和目录发生变化时应用程序可以捕获的事件

本章还将介绍System.IO.Compression名称空间，它允许读写压缩文件。我们主要看以下两个流类：

- DeflateStream——表示在写入时自动压缩数据或在读取时自动解压缩的流，使用Deflate算法来实现压缩。
- GZipStream——表示在写入时自动压缩数据或在读取时自动解压缩的流，使用GZIP（GNU Zip）算法来实现压缩。

18.1.1 File类和Directory类

File和Directory实用类提供了许多静态方法，用于处理文件和目录。这些方法可以移动文件、查询和更新特性，还可以创建FileStream对象。如第8章所述，可以在类上调用静态方法，而不必创建它们的实例。

File类的一些最常用的静态方法如表18-2所示。

表18-2 File类的静态方法

方法	说明
Copy()	将文件从源位置复制到目标位置
Create()	在指定的路径上创建文件

Delete()	删除文件
Open()	返回指定路径上的FileStream对象
Move()	将指定的文件移到新位置。可在新位置为文件指定不同名称

Directory类的一些常用静态方法如表18-3所示。

表18-3 Directory类的静态方法

方法	说明
CreateDirectory()	创建具有指定路径的目录
Delete()	删除指定的目录及其中的所有文件
GetDirectories()	返回表示指定目录下的目录名的string对象数组
EnumerateDirectories()	与GetDirectories()类似，但返回目录名的IEnumerable<string>集合
GetFiles()	返回在指定目录中的文件名的string对象数组
EnumerateFiles()	与GetFiles()类似，但返回文件名的IEnumerable<string>集合
GetFileSystemEntries()	返回指定目录中的文件和目录名的string对象数组
EnumerateFileSystemEntries()	与GetFileSystemEntries()类似，但返回文件和目录名的IEnumerable<string>集合

Move()	将指定目录移到新位置。可在新位置为文件夹指定一个新名称
--------	-----------------------------

其中的3个EnumerateXxx ()方法在存在大量文件或目录时，其性能比对应的GetXxx ()方法好。

18.1.2 FileInfo类

FileInfo类不像File类，它不是静态的，没有静态方法，只有在实例化后才可使用。FileInfo对象表示磁盘或网络位置上的文件。提供文件路径，就可以创建一个FileInfo对象：

```
FileInfo aFile = new FileInfo(@"C:\Log.txt");
```

注意： 本章处理的是表示文件路径的字符串，该字符串中有许多“\”字符，所以上述字符串的前缀@表示这个字符串应按字面意义解释，“\”解释为“\\”，而不解释为转义字符。如果没有@前缀，就需要用“\\”替代“\”，以免把这个字符解释为转义字符。本章总是在字符串前面加上前缀@。

也可以将目录名传送给FileInfo构造函数，但实际上这并不是很有用。这么做会用所有的目录信息初始化FileInfo的基类FileSystemInfo，但FileInfo中与文件相关的专用方法或属性都不会工作。

FileInfo类提供的许多方法类似于File类的方法，但由于File是静态

类，它需要一个字符串参数为每个方法调用指定文件位置。因此，下面的调用可以完成相同的工作：

```
FileInfo aFile = new FileInfo("Data.txt");
if (aFile.Exists)
    WriteLine("File Exists");
if (File.Exists("Data.txt"))
    WriteLine("File Exists");
```

这段代码检查文件Data.txt是否存在。注意，这里没有指定任何目录信息，这说明只检查当前的工作目录。这个目录包含调用此代码的应用程序。本章后面的“路径名和相对路径”一节将详细介绍这一内容。

FileInfo类的许多方法与File类中的对应方法类似。大多数情况下使用什么技术并不重要，但下面的规则有助于确定哪种技术更合适：

- 如果仅进行单一方法调用，则可以使用静态File类上的方法。在此，单一调用要快一些，因为.NET Framework不必实例化新对象，再调用方法。
- 如果应用程序在文件上执行几种操作，则实例化FileInfo对象并使用其方法就更好一些。这节省时间，因为对象已在文件系统上引用正确的文件，而静态类必须每次都寻找文件。

FileInfo类也提供了与底层文件相关的属性，其中一些属性可以用来更新文件，其中很多属性都继承于FileSystemInfo，所以可应用于FileInfo和DirectoryInfo类。FileSystemInfo类的属性如表18-4所示。

表18-4 **FileSystem**的属性

--	--

属性	说明
Attributes	使用FileAttributes枚举，获取或者设置当前文件或目录的特性
CreationTime, CreationTimeUtc	获取当前文件的创建日期和时间，可使用UTC和非UTC版本
Extension	提取文件的扩展名。这个属性是只读的
Exists	确定文件是否存在，这是一个只读的抽象属性，在FileInfo和DirectoryInfo中进行了重写
FullName	检索文件的完整路径，这个属性是只读的
LastAccessTime, LastAccessTimeUtc	获取或设置上次访问当前文件的日期和时间，可使用UTC和非UTC版本
LastWriteTime, LastWritetimeUtc	获取或设置上次写入当前文件的日期和时间，可使用UTC和非UTC版本
Name	检索文件的完整路径，这是一个只读抽象属性，在FileInfo和DirectoryInfo中进行了重写

FileInfo的专用属性如表18-5所示。

表18-5 FileInfo的属性

属性	说明
Directory	检索一个DirectoryInfo对象，表示包含当前文件的目录。这个属性是只读的
DirectoryName	返回文件目录的路径。这个属性是只读的
	文件只读特性的快捷方式。也可以通过Attributes来

IsReadOnly	访问这个属性
Length	获取文件的大小（以字节为单位），返回long值。这个属性是只读的

18.1.3 DirectoryInfo类

DirectoryInfo类的作用类似于FileInfo类。它是一个实例化的对象，表示计算机上的单一目录。与FileInfo类一样，在Directory和DirectoryInfo之间存在许多类似的方法调用。选择使用File或FileInfo方法的规则也适用于DirectoryInfo方法：

- 如果执行单一调用，就使用静态Directory类。
- 如果执行一系列调用，则使用实例化的DirectoryInfo对象。

DirectoryInfo类的大多数属性继承自FileSystemInfo，与FileInfo类一样，但这些属性作用于目录上，而不是文件上。还有两个DirectoryInfo专用属性，如表18-6所示。

表18-6 DirectoryInfo类的专用属性

属性	说明
Parent	检索一个DirectoryInfo对象，表示包含当前目录的目录。这个属性是只读的
Root	检索一个DirectoryInfo对象，表示包含当前目录的根目录，例如C:\目录。这个属性是只读的

18.1.4 路径名和相对路径

在.NET代码中指定路径名时，可使用绝对路径名，也可以使用相对路径名。绝对路径名显式地指定文件或目录来自于哪一个已知的位置，比如C:驱动器。它的一个示例是C:\Work\LogFile.txt。注意这个路径准确地定义了其位置。

相对路径名相对于一个起始位置。使用相对路径名时，不必指定驱动器或已知的位置；前面的当前工作目录就是起点，这是相对路径名的默认设置。例如，如果应用程序运行在C:\Development\FileDemo目录上，并使用相对路径LogFile.txt，该文件就是C:\Development\FileDemo\LogFile.txt。为上移目录，要使用..字符串。这样，在同一个应用程序中，路径..\Log.txt表示C:\Development\Log.txt文件。

如前所述，工作目录起初设置为运行应用程序的目录。当使用VS开发程序时，这就表示应用程序是所创建的项目文件夹下的几个目录。它通常位于*ProjectName* \bin\Debug。要访问项目根文件夹中的文件，必须用..\..\上移两个目录，这在本章中十分常见。

如有必要，可使用Directory.GetCurrentDirectory()找出工作目录的当前设置，也可以使用Directory.SetCurrentDirectory()设置新路径。

18.2 流

在.NET Framework中进行的所有输入和输出工作都要用到流（stream）。流是序列化设备（serial device）的抽象表示。序列化设备可以线性方式存储数据，并可按同样的方式访问：一次访问一个字节。此设备可以是磁盘文件、网络通道、内存位置或其他支持以线性方式读写的对象。把设备变成抽象的，就可以隐藏流的底层目标和源。这种抽象级别支持代码重用，允许编写更通用的例程，因为不必担心数据传输方式的特性。因此，当应用程序从文件输入流、网络输入流或其他流中读取数据时，就可以传输和重用类似的代码。而且，使用文件流还可以忽略每种设备的物理机制，不必担心硬盘磁头或内存分配问题。

流可以表示几乎所有源，例如键盘、物理磁盘文件、网络位置、打印机。甚至另一个程序，但本章仅关注磁盘文件的读写。适用于读写磁盘文件的概念，也适用于大多数设备，所以读者可以基本理解流的概念，学习可用于许多情形的、已证明有效的方法。

18.2.1 使用流的类

使用流的类，与File和Directory类一样，也包含在System.IO名称空间中。这些类如表18-7所示。

表18-7 流类

类	说明
---	----

FileStream	表示可写或可读，或二者均可的文件。可以同步或异步地读写此文件
StreamReader	从流中读取字符数据，可以使用FileStream作为基类创建
StreamWriter	向流写入字符数据，可以使用FileStream作为基类创建

下面看看如何使用这些类。

18.2.2 FileStream对象

FileStream对象表示指向磁盘或网络路径上的文件的流。这个类提供了在文件中读写字节的方法，但经常使用StreamReader或StreamWriter执行这些功能。这是因为FileStream类操作的是字节和字节数组，而Stream类操作的是字符数据。字符数据易于使用，但是有些操作，如随机文件访问（访问文件中间某点的数据），就必须由FileStream对象执行，稍后对此进行介绍。

还有几种方法可以创建FileStream对象。其构造函数具有许多不同的重载版本，最简单的构造函数仅有两个参数，即文件名和FileMode枚举值。

```
FileStream aFile = new FileStream(filename, FileMode.<Member  
  
>);
```

FileMode枚举包含几个成员，指定了如何打开或创建文件。稍后介绍这些枚举成员。另一个常用的构造函数如下：

```
FileStream aFile =
    new FileStream(filename, FileMode.<Member
>, FileAccess.<Member
>);
```

第三个参数是FileAccess枚举的一个成员，它指定了流的作用。FileAccess枚举的成员如表18-8所示。

表18-8 FileMode枚举成员

成员	说明
Read	打开文件，用于只读
Write	打开文件，用于只写
ReadWrite	打开文件，用于读写

对文件进行非FileAccess枚举成员指定的操作会导致抛出异常。此属性的作用是，基于用户的权限级别改变用户对文件的访问权限。

在FileStream构造函数不使用FileAccess枚举参数的版本中，使用默认值FileAccess.ReadWrite。

FileMode枚举成员如表18-9所示。使用每个值会发生什么，取决于指定的文件名是否表示已有的文件。注意，这个表中的项表示创建流时该流指向文件中的位置，下一节将详细讨论这个主题。除非特别说明，否则流就指向文件的开头处。

表18-9 FileMode枚举成员

成员	文件存在	文件不存在
Append	打开文件，流指向文件的末尾处，只能与枚举FileAccess.Write结合使用	创建一个新文件。只能与枚举FileAccess.Write结合使用
Create	删除该文件，然后创建新文件	创建新文件
CreateNew	抛出异常	创建新文件
Open	打开文件，流指向文件开头处	抛出异常
OpenOrCreate	打开文件，流指向文件开头处	创建新文件
Truncate	打开文件，清除其内容。流指向文件开头处，保留文件的初始创建日期	抛出异常

File和FileInfo类都提供了OpenRead()和OpenWrite()方法，更易于创建FileStream对象。前者打开了只读访问的文件，后者只允许写入文件。这些都提供了快捷方式，因此不必以FileStream构造函数的参数形式提供所有必要的信息。例如，下面的代码行打开了用于只读访问的Data.txt文件：

```
FileStream aFile = File.OpenRead("Data.txt");
```

下面的代码执行同样的功能：

```
FileInfo aFileInfo = new FileInfo("Data.txt");  
FileStream aFile = aFileInfo.OpenRead();
```

1. 文件位置

`FileStream`类维护内部文件指针，该指针指向文件中进行下一次读写操作的位置。大多数情况下，当打开文件时，它就指向文件的开始位置，但是可以修改此指针。这允许应用程序在文件的任何位置读写，随机访问文件，或直接跳到文件的特定位置上。当处理大型文件时，这非常省时，因为马上就可以找到正确位置。

实现此功能的方法是`Seek()`方法，它有两个参数：第一个参数指定文件指针移动距离（以字节为单位）。第二个参数指定开始计算的起始位置，用`SeekOrigin`枚举的一个值表示。`SeekOrigin`枚举包含3个值：`Begin`、`Current`和`End`。

例如，下面的代码行将文件指针移到文件的第8个字节处，其起始位置就是文件的第1个字节：

```
aFile.Seek(8, SeekOrigin.Begin);
```

下面的代码行将文件指针从当前位置开始向前移动2个字节。如果在上面的代码行之后执行下面的代码，文件指针就指向文件的第10个字节：

```
aFile.Seek(2, SeekOrigin.Current);
```

注意读写文件时，文件指针会随之改变。在读取了10个字节之后，文件指针就指向被读取的第10个字节之后的字节。

也可以指定负查找位置，这可与SeekOrigin.End枚举值一起使用，查找靠近文件末端的位置。下面的代码会查找文件中倒数第5个字节：

```
aFile.Seek(-5, SeekOrigin.End);
```

采用这种方式访问的文件有时称为随机访问文件，因为应用程序可以访问文件中的任何位置。稍后介绍的StreamReader和StreamWriter类可以连续地访问文件，但不允许以这种方式操作文件指针。

2. 读取数据

使用FileStream类读取数据不像使用本章后面介绍的StreamReader类读取数据那样容易。这是因为FileStream类只能处理原始字节（raw byte）。处理原始字节的功能使FileStream类可以用于任何数据文件，而不仅仅是文本文件。通过读取字节数据，FileStream对象可以用于读取诸如图像和声音的文件。这种灵活性的代价是，不能使用FileStream类将数据直接读入字符串，而使用StreamReader类却可以这样处理。但是有几种转换类可以很轻易地将字节数组转换为字符数组，或者将字符数组转换为字节数组。

FileStream.Read()方法是从FileStream对象所指向的文件中访问数据的主要手段。这个方法从文件中读取数据，再把数据写入一个字节数组。它有三个参数：第一个参数是传入的字节数组，用来接受

FileStream对象中的数据。第二个参数是字节数组中开始写入数据的位置；它通常是0，表示从数组开端向文件中写入数据。最后一个参数指定从文件中读出多少字节。

下面的示例演示了从随机访问文件中读取数据。要读取的文件实际是为此示例创建的分类文件。

试一试：从随机访问文件中读取数据：**ReadFile\Program.cs**

(1) 在C:\BegVCSharp\Chapter18目录中创建一个新的控制台应用程序ReadFile。

(2) 在Program.cs文件的顶部添加下面的using指令：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
```

(3) 在Main()方法中添加以下代码：

```
static void Main(string[] args)
{
    byte[] byteData = new byte[200];
```

```
char[] charData = new char[200];
```

```
try
```

```
{
```

```
    FileStream aFile = new FileStream("../..//Program.cs", Fi
```

```
    aFile.Seek(174, SeekOrigin.Begin);
```

```
    aFile.Read(byteData, 0, 200);
```

```
}
```

```
        catch(IOException e)

        {

            WriteLine("An IO exception has been thrown!");
            WriteLine(e.ToString());
            ReadKey();
            return;
        }
        Decoder d = Encoding.UTF8.GetDecoder();
        d.GetChars(byteData, 0, byteData.Length, charData, 0);
        WriteLine(charData);
        ReadKey();
    }
```

(4) 运行应用程序。结果如图18-1所示。

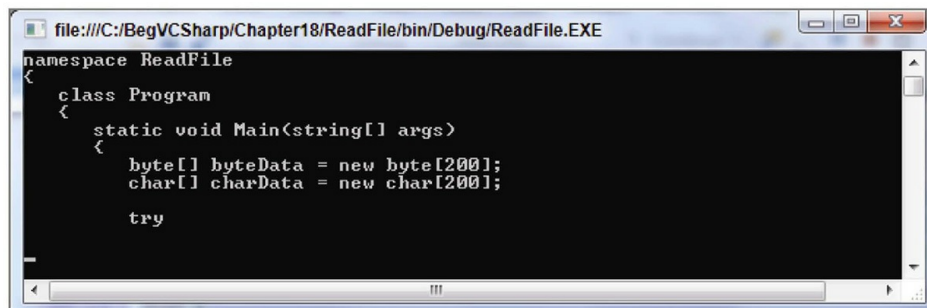


图18-1

示例的说明

此应用程序打开自己的.cs文件，用于从中读取。它在下面的代码行中使用..字符串向上逐级导航两个目录，找到该文件：

```
FileStream aFile = new FileStream("../../Program.cs", FileMode.Open);
```

下面两行代码执行查找工作，并从文件的具体位置读取字节：

```
aFile.Seek(174, SeekOrigin.Begin);
aFile.Read(byteData, 0, 200);
```

第一行代码将文件指针移到文件的第174个字节。在Program.cs文件中，是namespace中的“n”；其前面的174个字符是using指令。第二行将接下来的200个字节读入到byteData字节数组中。

注意，这两行代码封装在try...catch块中，以便处理可能抛出的异常。

```
try
{
```

```
aFile.Seek(113, SeekOrigin.Begin);
aFile.Read(byteData, 0, 100);
}
catch(IOException e)
{
    WriteLine("An IO exception has been thrown!");
    WriteLine(e.ToString());
    ReadKey();
    return;
}
```

文件I/O涉及的所有操作几乎都可以抛出IOException类型的异常。所有产品代码都必须包含错误处理，在处理文件系统时尤其如此。本章的所有示例都包含基本的错误处理代码。

从文件中获取了字节数组后，就需要将其转换为字符数组，以便在控制台显示它。为此，使用System.Text名称空间的Decoder类。此类用于将原始字节转换为更有用的项，比如字符：

```
Decoder d = Encoding.UTF8.GetDecoder();
d.GetChars(byteData, 0, byteData.Length, charData, 0);
```

这些代码基于UTF-8编码模式创建了Decoder对象。这就是Unicode编码模式。然后调用GetChars()方法，此方法接受一个字节数组作为参数，将其转换为字符数组。完成后，就可以将字符数组输出到控制台。

3. 写入数据

向随机访问文件中写入数据的过程与从中读取数据非常类似。首先需要创建一个字节数组；最简单的办法是首先构建要写入文件的字符数组。然后使用Encoder对象将其转换为字节数组，其用法非常类似于Decoder对象。最后调用Write()方法，将字节数组传送到文件中。

下面构建一个简单的示例演示其过程。

试一试：将数据写入随机访问文件：**WriteFile\Program.cs**

(1) 在C:\BegVCSharp\Chapter18目录中创建一个新的控制台应用程序WriteFile。

(2) 在Program.cs文件顶部添加下面的using指令：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
```

(3) 在Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
```

```
byte[] byteData;
```

```
char[] charData;
```

```
try
```

```
{
```

```
    FileStream aFile = new FileStream("Temp.txt", FileMode.Cre
```

```
    charData = "My pink half of the drainpipe.".ToCharArray();
```

```
    byteData = new byte[charData.Length];
```

```
Encoder e = Encoding.UTF8.GetEncoder();
```

```
e.GetBytes(charData, 0, charData.Length, byteData, 0, true
```

```
// Move file pointer to beginning of file.
```

```
aFile.Seek(0, SeekOrigin.Begin);
```

```
aFile.Write(byteData, 0, byteData.Length);
```

```
}
```

```
catch (IOException ex)
```

```
{  
  
    WriteLine("An IO exception has been thrown!");  
    WriteLine(ex.ToString());  
    ReadKey();  
    return;  
  
}  
  
}
```

(4) 运行该应用程序。它在短暂运行后将会关闭。

(5) 导航到应用程序目录——在目录中已经保存了文件，因为我们使用了相对路径。目录位于WriteFile\bin\Debug文件夹中。打开Temp.txt文件。可在文件中看到如图18-2所示的文本。

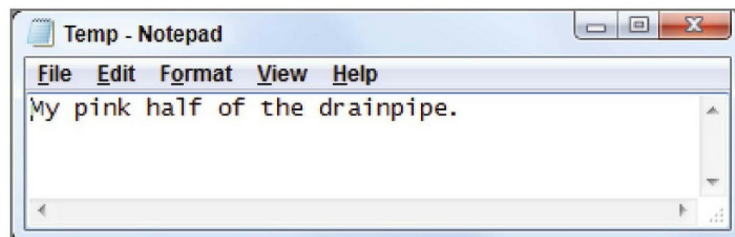


图18-2

示例的说明

此应用程序打开自己的目录中的文件，并在文件中写入了一个简单字符串。这个示例的结构非常类似于前面的示例，只是用Write()替代了Read()，用Encoder替代了Decoder。

下面的代码行使用String类的ToCharArray()方法，创建了字符数组。因为C#中的所有事物都是对象，文本“My pink half of the drainpipe.”实际上是一个String对象（尽管有点儿怪），所以甚至可在字符串上调用这些静态方法。

```
CharData = "My pink half of the drainpipe.".ToCharArray();
```

下面的代码行显示了如何将字符数组转换为FileStream对象需要的正确字节数组。

```
Encoder e = Encoding.UTF8.GetEncoder();  
e.GetBytes(charData, 0, charData.Length, byteData, 0, true);
```

这次，要基于UTF-8编码方法来创建Encoder对象。也可将Unicode用于解码，此时在写入流之前，需要将字符数据编码为正确的字节格式。在GetBytes()方法中可以完成这些工作，它可将字符数组转换为字

节数组。GetByte()方法将字符数组作为第一个参数（本例中的charData），将该数组中起始位置的下标作为第二个参数（0表示数组的开头）。第三个参数是要转换的字符数量（charData.Length，charData数组中的元素个数）。第四个参数是在其中放入数据的字节数组（byteData），第五个参数是在字节数组中开始写入位置的索引（0表示byteData数组的开头）。

最后一个参数决定在结束后Encoder对象是否应该更新其状态，这反映了一个事实：Encoder对象在内存中保持记录它原来在字节数组中的位置。这有助于以后调用Encoder对象，但是当只调用一次时，这就没什么意义。最后对Encoder的调用必须将此参数设置为true，以清空其内存，释放对象，用于垃圾回收。

此后使用Write()方法向FileStream写入字节数组就变得非常简单：

```
aFile.Seek(0, SeekOrigin.Begin);  
aFile.Write(byteData, 0, byteData.Length);
```

与Read()方法一样，Write()方法也有三个参数：包含要写入文件流的数据的字节数组，开始写入的数组索引和要写入的字节数。

18.2.3 StreamWriter对象

操作字节数组比较麻烦，因为使用FileStream对象非常困难，那么，还有简单一些的方法吗？答案是有的，因为有了FileStream对象，通常会创建一个StreamWriter或StreamReader，并使用它们的方法来处理文件。如果不需要将文件指针改变到任意位置，使用这些类就很容易操

作文件。

`StreamWriter`类允许将字符和字符串写入到文件中，它处理底层的转换，向`FileStream`对象写入数据。

还可以通过许多方法创建`StreamWriter`对象。如果已经有了`FileStream`对象，则可以使用此对象来创建`StreamWriter`对象：

```
FileStream aFile = new FileStream("Log.txt", FileMode.CreateNe  
StreamWriter sw = new StreamWriter(aFile);
```

也可以直接从文件中创建`StreamWriter`对象：

```
StreamWriter sw = new StreamWriter("Log.txt", true);
```

这个构造函数的参数是文件名和一个`Boolean`值，这个`Boolean`值指定是追加文件，还是创建新文件：

- 如果此值设置为`false`，则创建一个新文件，或者截取现有文件并打开它。
- 如果此值设置为`true`，则打开文件，保留原来的数据。如果找不到文件，则创建一个新文件。

与创建`FileStream`对象不同，创建`StreamWriter`对象不会提供一组类似的选项：除了使用`Boolean`值追加文件或创建新文件外，根本没有像`FileStream`类那样指定`FileMode`属性的选项。而且，没有设置`FileAccess`属性的选项，因此总是拥有对文件的读/写权限。为使用高级参数，必须首先在`FileStream`构造函数中指定这些参数，然后在`FileStream`对象中创建`StreamWriter`，如下面的示例所示。



试一试：将数据写入输出流： **StreamWrite\Program.cs**

(1) 在C:\BegVCSharp\Chapter18目录中创建一个新的控制台应用程序StreamWrite。

(2) 再次使用System.IO名称空间，因此在Program.cs文件靠近顶部的位置添加下面的using指令：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
```

(3) 在Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
    try
    {
        FileStream aFile = new FileStream("Log.txt", FileMode.Open)
        StreamWriter sw = new StreamWriter(aFile);
```



```
bool truth = true;
// Write data to file.
sw.WriteLine("Hello to you.");
sw.Write($"It is now {DateTime.Now.ToLongDateString()}");
sw.Write("and things are looking good.");
sw.Write("More than that,");
sw.Write($" it's {truth} that C# is fun.");
sw.Close();
}
catch(IOException e)
{
    WriteLine("An IO exception has been thrown!");
    WriteLine(e.ToString());
    ReadLine();
    return;
}
}
```

(4) 生成并运行该项目。如果没有错误，则项目会很快运行并关闭。因为我们在控制台上没有显示任何内容，所以在控制台中看不到程序的执行情况。

(5) 进入应用程序目录，找到Log.txt文件，它位于StreamWrite\bin\Debug文件夹中，这是因为我们使用了相对路径。

(6) 打开文件，可以看到图18-3所示的文本。

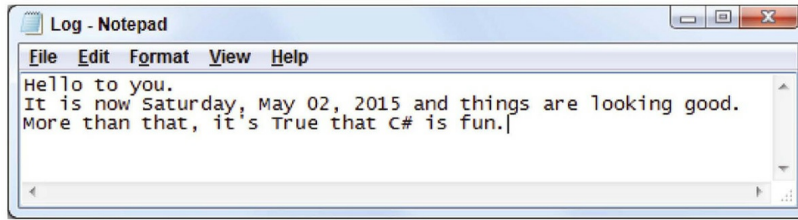


图18-3

示例的说明

这个简单的应用程序演示了StreamWriter类的两个最重要方法：Write()和WriteLine()。这两个方法具有许多重载的版本，可以完成更高级的文件输出，但本示例只使用基本的字符串输出。

WriteLine()方法会写入传递给它的字符串，其后跟有换行符。在此示例中可以看到，下一个写入操作在新行上开始。

```
sw.WriteLine("Hello to you.");
```

Write()方法只是把传给它的字符串写入文件，但不追加换行符，因此可使用多个Write()语句写入完整的句子或段落。如同可以向控制台写入格式化数据一样，也可以向文件写入格式化数据。例如，可使用标准格式化参数把变量的值写入文件：

```
sw.Write($"It is now {DateTime.Now.ToLongDateString()}");
```

DateTime.Now存储当前日期，ToLongDateString()方法用于把这个日期转换为便于读取的格式。

```
sw.Write("More than that, ");  
sw.Write(" it's {truth} that C# is fun.");
```

这里也使用了格式化参数，这次使用Write()显示布尔值truth。前面把这个变量设置为true，其值会自动格式化，转换为字符串“True”。

可使用Write()和格式化参数写入用逗号分隔的文件：

[StreamWriter object

```
].Write("${100},{\"A nice product\"},{10.50}\");
```

在更复杂的示例中，这些数据还可以来自数据库或其他数据源。

18.2.4 StreamReader对象

输入流用于从外部源中读取数据。很多情况下，数据源是磁盘上的文件或网络的某些位置。任何可以发送数据的位置都可以是数据源，比如网络应用程序，甚至是控制台。

用来从文件中读取数据的类是StreamReader。与StreamWriter一样，这是一个通用类，可以用于任何流。下面的示例会再次围绕FileStream对象构造StreamReader类，使其指向正确的文件。

StreamReader对象的创建方式与StreamWriter对象非常类似。创建它的最常见方式是使用前面创建的FileStream对象：

```
FileStream aFile = new FileStream("Log.txt", FileMode.Open);  
StreamReader sr = new StreamReader(aFile);
```

与StreamWriter一样，可以直接用包含具体文件路径的字符串创建StreamReader类：

```
StreamReader sr = new StreamReader("Log.txt");
```

试一试：从输入流中读取数据：**StreamRead\Program.cs**

（1）在C:\BegVCSharp\Chapter18目录中创建一个新控制台应用程序StreamRead。

（2）必须再次导入System.IO和System.Console名称空间，因此将下面的代码放在Program.cs文件的靠近顶部的位置：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using static System.Console;
```

（3）在Main()方法中添加下面的代码：

```
static void Main(string[] args)
{
    string line;
```

```
try
```

```
{
```

```
    FileStream aFile = new FileStream("Log.txt", FileMode.Open);
```

```
    StreamReader sr = new StreamReader(aFile);
```

```
    line = sr.ReadLine();
```

```
    // Read data in line by line.
```

```
    while(line != null)
```

```

{

    WriteLine(line);
    line = sr.ReadLine();
}
sr.Close();
}
catch(IOException e)
{
    WriteLine("An IO exception has been thrown!");
    WriteLine(e.ToString());
    return;
}
ReadKey();
}

```

(4) 把前面示例中创建的Log.txt文件复制到StreamRead\bin\Debug目录中。如果没有Log.txt文件，FileStream构造函数找不到该文件，就会抛出异常。

(5) 运行该应用程序，可以看到写入到控制台的文件文本，如图18-4所示。

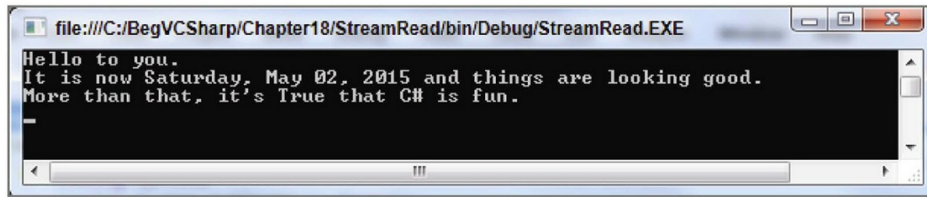


图18-4

示例的说明

这个应用程序与前面的应用程序非常类似。其明显区别就是，它是在读取数据，而不是写入数据。与前面一样，只有导入System.IO名称空间，才能访问需要的类。

使用ReadLine()方法从文件中读取文本。这个方法读取换行符之前的文本，并以字符串的形式返回结果文本。当到达文件尾时，该方法就返回空值，通过这种方法可以测试文件是否已到达了尾部。注意使用了while循环，以便确保在执行循环体的代码之前读取的行不为空，这样就只显示文件的有效内容：

```
line = sr.ReadLine();
while(line != null)
{
    WriteLine(line);
    line = sr.ReadLine();
}
```

读取数据

`ReadLine()`方法不是在文件中访问数据的唯一方法。`StreamReader`类还包含许多读取数据的方法。

读取数据最简单的方法是`Read()`。此方法将流的下一个字符作为正整数值返回，如果到达了流的结尾处，则返回-1。使用`Convert`实用类可以把这个值转换为字符。在上面的示例中，可以按如下方式重新编写程序的主体：

```
StreamReader sr = new StreamReader(aFile);  
int charCode;
```

```
charCode = sr.Read();
```

```
while(charCode != -1)
```

```
{
```

```
Write(Convert.ToChar(charCode));
```



```
        charCode = sr.Read();  
  
    }
```

```
sr.Close();
```

对于小型文件，可使用一个非常简便的方法ReadToEnd()。此方法读取整个文件，并将其作为字符串返回。在此，前面的应用程序可以简化为：

```
StreamReader sr = new StreamReader(aFile);  
line = sr.ReadToEnd();
```

```
WriteLine(line);
```

```
sr.Close();
```

这似乎非常简便，但必须小心。将所有数据读取到字符串对象中，

会迫使文件中的数据放到内存中。应根据数据文件的大小禁止这样处理。如果数据文件非常大，最好将数据留在文件中，并使用 `StreamReader` 的方法访问文件。

处理大型文件的另一种方式是 .NET 4 中新增的静态方法 `File.ReadLines()`。实际上，`File` 的几个静态方法可用于简化文件数据的读写，但这个方法特别有趣，因为它返回 `IEnumerable<string>` 集合。可以迭代这个集合中的字符串，一次读取文件中的一行。使用这个方法，将前面的示例重写为：

```
foreach (string alternativeLine in File.ReadLines("Log.txt"))
    WriteLine(alternativeLine);
```

可以看出，在 .NET 中，可通过多种不同的方式获得相同的结果——读取文件中的数据。可以选择其中最适当的技术。

18.2.5 异步文件访问

有时，例如要一次性执行大量文件访问操作，或者要处理非常大的文件，读写文件系统数据是很缓慢的。此时，你可能想在等待这些操作完成的同时执行其他操作。这对于桌面应用程序尤为重要，因为在桌面应用程序中，需要让应用程序在后台进行处理的同时，对用户保持良好的响应性。

为帮助实现这种操作，.NET 4.5 引入了一些新的异步方式来操作流。这种异步方式适用于 `FileStream` 类，也适用于 `StreamReader` 类和 `StreamWriter` 类。如果查看这些类的定义，可找到带有 `Async` 后缀的方

法，例如StreamReader类的ReadLineAsync()方法，它是ReadLine()方法的异步版本。这些方法在新的基于任务的异步编程模型中使用。

异步编程是一种高级技术，本书中不详细讨论。但是，如果你对异步文件系统访问感兴趣，可以把这里介绍的内容作为起点。更多信息可以阅读Christian Nagel、Jay Glynn和Morgan Skinner撰写的《C#高级编程（第9版）》（Wrox, 2014）。

18.2.6 读写压缩文件

在处理文件时，常会占用大量硬盘空间。图形和声音文件尤其如此。读者可能使用过能压缩文件和解压文件的工具，当希望带着文件到其他地方去或者通过电子邮件把文件发送给他人时，使用这些工具是很方便的。System.IO.Compression名称空间就包含能在代码中压缩文件的类，这些类使用GZIP或Deflate算法，这两种算法都是公开的、免费的，任何人都可以使用。

但压缩文件并不只是把它们压缩一下就完事了。商业应用程序允许把多个文件放在一个压缩文件（通常称为存档文件）中。System.IO.Compression名称空间中的一些类提供了类似功能。但为了简洁起见，本节介绍的内容要简单得多：只是把文本数据保存在压缩文件中。不能在外部实用程序中访问这个文件，但这个文件比未压缩版本要小得多。

System.IO.Compression名称空间中有两个压缩流类DeflateStream和GZipStream，它们的工作方式非常类似。对于这两个类，都要用已有的流初始化它们，对于文件，流就是FileStream对象。此后就可以把它们

用于StreamReader和StreamWriter了，就像使用其他流一样。此外，只需要指定流是用于压缩（保存文件）还是解压缩（加载文件），类就要对传送给它的数据执行什么操作。这最好用一个示例来加以说明。

试一试：读写压缩数据：Compressor\Program.cs

（1）在C:\BegVCSharp\Chapter18目录中创建一个新的控制台应用程序Compressor。

（2）将下面的代码放在Program.cs靠近顶部的位置。只有导入System.IO、System.Console和System.IO.Compression名称空间才能使用文件和压缩类：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

using System.IO.Compression;
```

```
using static System.Console;
```

(3) 把下面的方法添加到Program.cs中Main()方法的前面:

```
static void SaveCompressedFile(string filename, string data)
{
    FileStream fileStream =
        new FileStream(filename, FileMode.Create, FileAccess.Write);
    GZipStream compressionStream =
        new GZipStream(fileStream, CompressionMode.Compress);
    StreamWriter writer = new StreamWriter(compressionStream);
    writer.Write(data);
    writer.Close();
}

static string LoadCompressedFile(string filename)
{
    FileStream fileStream =
        new FileStream(filename, FileMode.Open, FileAccess.Read);
    GZipStream compressionStream =
        new GZipStream(fileStream, CompressionMode.Decompress);
    StreamReader reader = new StreamReader(compressionStream);
    string data = reader.ReadToEnd();
    reader.Close();
    return data;
}
```

```
}
```

(4) 为Main()方法添加如下代码:

```
static void Main(string[] args)
```

```
{
```

```
    try
```

```
{
```

```
    string filename = "compressedFile.txt";
```

```
    WriteLine(
```

```
        "Enter a string to compress (will be repeated 100 times)
```

```
        string sourceString = ReadLine());
```

```
StringBuilder sourceStringMultiplier =  
  
    new StringBuilder(sourceString.Length * 100);  
  
    for (int i = 0; i<100; i++)  
  
    {  
  
        sourceStringMultiplier.Append(sourceString);  
  
    }
```

```
sourceString = sourceStringMultiplier.ToString();
```

```
WriteLine($"Source data is {sourceString.Length} bytes lon
```

```
SaveCompressedFile(filename, sourceString);
```

```
WriteLine($"\\nData saved to {filename}.");
```

```
FileInfo compressedFileData = new FileInfo(filename);
```

```
Write($"Compressed file is {compressedFileData.Length}");
```

```
WriteLine(" bytes long.");
```



```
string recoveredString = LoadCompressedFile(filename);
```

```
recoveredString = recoveredString.Substring(
```

```
0, recoveredString.Length / 100);
```

```
WriteLine($"\\nRecovered data: {recoveredString}",);
```

```
ReadKey();
```

```
}
```

```
catch (IOException ex)
```

```
{  
  
    WriteLine("An IO exception has been thrown!");  
  
    WriteLine(ex.ToString());  
  
    ReadKey();  
  
}  
  
}
```

(5) 运行应用程序，输入一个长度合理的长字符串，结果如图18-5所示。

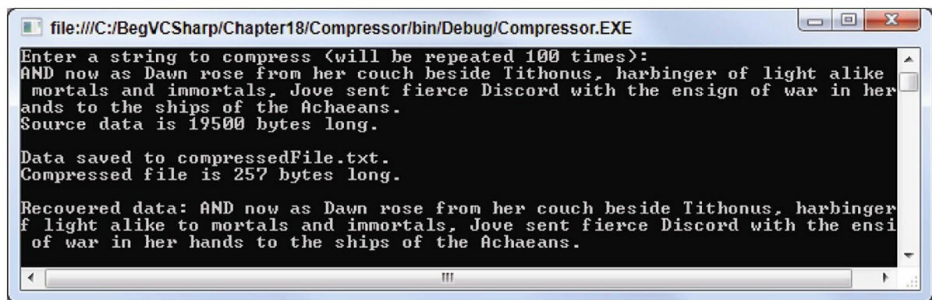


图18-5

(6) 在记事本中打开compressedFile.txt，文本如图18-6所示。

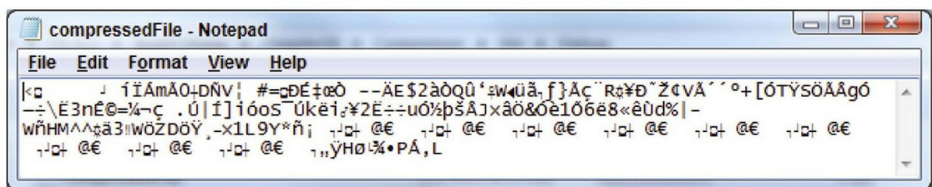


图18-6

示例的说明

这个示例定义了两个方法，用于保存和加载已压缩的文本文件。第一个方法是Save CompressedFile()，如下所示：

```
static void SaveCompressedFile(string filename, string data)
{
    FileStream fileStream =
        new FileStream(filename, FileMode.Create, FileAccess.Write);
    GZipStream compressionStream =
        new GZipStream(fileStream, CompressionMode.Compress);
    StreamWriter writer = new StreamWriter(compressionStream);
```

```
        writer.Write(data);  
        writer.Close();  
    }
```

代码首先创建一个`FileStream`对象，然后使用它创建一个`GZipStream`对象。注意，可以用`DeflateStream`替换这段代码中的所有`GZipStream`——这两个类的工作方式相同。使用`CompressionMode.Compress`枚举值指定数据要进行压缩，然后使用`StreamWriter`将数据写入文件。

`LoadCompressedFile()`方法与`SaveCompressedFile()`方法正好相对，它不是保存到文件名中，而是把压缩的文件加载到字符串中：

```
static string LoadCompressedFile(string filename)  
{  
    FileStream fileStream =  
        new FileStream(filename, FileMode.Open, FileAccess.Read);  
    GZipStream compressionStream =  
        new GZipStream(fileStream, CompressionMode.Decompress);  
    StreamReader reader = new StreamReader(compressionStream);  
    string data = reader.ReadToEnd();  
    reader.Close();  
    return data;  
}
```

其区别很显然：使用了不同的`FileMode`、`FileAccess`和`CompressionMode`枚举值来加载和解压数据，使用`StreamReader`从文件中提取出未压缩的文本。

Main()中的代码是这些方法的一个简单测试。它请求一个字符串，将字符串复制100次，再把它压缩到一个文件中，之后检索它。在本示例中，把《伊利亚特》第6章的第一句重复100次就有19 400个字符，但压缩后只占用225个字节，压缩率是80:1。应该承认，这有以偏盖全之嫌，GZIP算法很适合重复数据，但这里仅演示压缩过程而已。

我们还查看了存储在压缩文件中的文本。显然，它的意义很难明白。在应用程序之间共享数据时要考虑到这一点。但是，因为是用已知的算法压缩文件，所以至少能够知道应用程序是可以解压缩它的。

18.3 监控文件系统

有时，应用程序所需要完成的工作不仅限于从文件系统中读写文件。例如，知道修改文件或目录的时间非常重要。.NET Framework允许方便地创建完成这些任务的定制应用程序。

帮助完成这些任务的类是FileSystemWatcher。这个类提供了几个应用程序可以捕获的事件。应用程序可以对文件系统事件作出响应。

使用FileSystemWatcher的基本过程非常简单。首先必须设置一些属性，指定监控的位置、内容以及引发应用程序要处理的事件的时间。然后给FileSystemWatcher提供定制事件处理程序的地址，当发生重要事件时，FileSystemWatcher就可以调用这些事件处理程序。最后打开FileSystemWatcher，等待事件。

在启用FileSystemWatcher对象之前必须设置的属性如表18-10所示。

表18-10 FileSystemWatcher的属性

属性	说明
Path	设置要监控的文件位置或目录
NotifyFilter	这是NotifyFilters枚举值的组合，NotifyFilters枚举值指定了在被监控的文件内要监控哪些内容。这些表示要监控的文件或文件夹的属性。如果指定的属性发生了变化，就引发事件。可能的枚举值是Attributes、CreationTime、DirectoryName、FileName、

	LastAccess、LastWrite、Security和Size。注意，可通过二元OR运算符来合并这些枚举值
Filter	指定要监控哪些文件的过滤器，例如， *.txt

设置之后，就必须为4个事件Changed、Created、Deleted和Renamed编写事件处理程序。如第13章所述，这需要创建自己的方法，并将方法赋给对象的事件。将自己的事件处理程序赋给这些方法，就可以在引发事件时调用方法。当修改与Path、NotifyFilter和Filter属性匹配的文件或目录时，就引发每个事件。

设置了属性和事件后，将EnableRaisingEvents属性设置为true，就可以开始监控工作。下面的示例将在一个简单的客户应用程序中使用FileSystemWatcher，来监控所选的目录。

试一试：监控文件系统：FileWatch

这是一个比较复杂的示例，使用了本章介绍的许多内容。

（1）在C:\BegVCSharp\Chapter18目录中创建一个新的WPF应用程序FileWatch。

（2）修改MainWindow.xaml，如下所示（图18-7显示了结果窗口）：

```
<Window x:Class="FileWatch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
```

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
Title="File Monitor
```

```
" Height="160
```

```
" Width="300
```

```
<Grid>
```

```
<Grid.RowDefinitions>
```

```
<RowDefinition Height="Auto" />
```

```
<RowDefinition Height="Auto" />
```

```
<RowDefinition />
```



```
</Grid.RowDefinitions>
```

```
<Grid Margin="4">
```

```
<Grid.ColumnDefinitions>
```

```
<ColumnDefinition />
```

```
<ColumnDefinition Width="Auto" />
```

```
</Grid.ColumnDefinitions>
```

```
<TextBox Name="LocationBox" TextChanged="LocationBox_Text1
```

```
<Button Name="BrowseButton" Grid.Column="1" Margin="4,0,(
```

```
Content="Browse..." Click="BrowseButton_Click" />
```

```
</Grid>
```

```
<Button Name="WatchButton" Content="Watch!" Margin="4" Gri
```

```
Click="WatchButton_Click" IsEnabled="False" />
```

```
<ListBox Name="WatchOutput" Margin="4" Grid.Row="2" />
```

```
</Grid>
```

</Window>

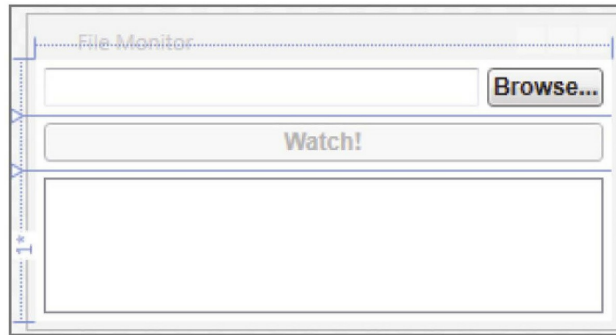


图18-7

（3）在MainWindow.xaml.cs中添加下面的using指令：

```
using System.IO;
```

```
using Microsoft.Win32;
```

（4）在MainWindow类中添加FileSystemWatcher类型的一个字段：

```
namespace FileWatch
{
    ///<summary>
    /// Interaction logic for MainWindow.xaml
    ///</summary>
```

```
public partial class MainWindow : Window
{
    // File System Watcher object.
```

```
private FileSystemWatcher watcher;
```

（5）在MainWindow类中添加下面的实用方法，以允许后台线程向输出添加消息：

```
private void AddMessage(string message
```

```
{
```

```
Dispatcher.BeginInvoke(new Action(
```

```
() => WatchOutput.Items.Insert(
```

```
0, message))));  
  
}
```

（6）在MainWindow类的构造函数的InitializeComponent()方法之后，添加下面的代码。这段代码用于初始化FileSystemWatcher对象，以及将事件关联到AddMessage()方法调用：

```
public MainWindow()  
{  
    InitializeComponent();  
    watcher = new FileSystemWatcher();  
  
    watcher.Deleted += (s, e) =>
```

```
AddMessage($"File: {e.FullPath} Deleted");
```

```
watcher.Renamed += (s, e) =>
```

```
AddMessage($"File renamed from {e.OldName} to {e.FullP
```

```
watcher.Changed += (s, e) =>
```

```
AddMessage($"File: {e.FullPath} {e.ChangeType.ToString
```

```
watcher.Created += (s, e) =>
```

```
AddMessage($"File: {e.FullPath} Created");
```

```
}
```

（7）添加Browse按钮的Click事件处理程序。这个事件处理程序中的代码会打开Open File对话框，供用户选择要监控的文件：

```
private void BrowseButton_Click(object sender, RoutedEventArgs)
```

```
{
```

```
    OpenFileDialog dialog = new OpenFileDialog();
```

```
    if (dialog.ShowDialog(this) == true)
```

```
    {
```

```
        LocationBox.Text = dialog.FileName;
```

```
}
```

```
}
```

ShowDialog()方法返回一个bool?值，反映用户退出File Open对话框的方式（用户可能单击OK按钮，或者单击Cancel按钮）。需要确认用户未单击Cancel按钮，所以在把用户选择的文件保存到TextBox中前，将ShowDialog()方法调用的结果与true进行比较。

（8）添加TextBox的事件处理程序TextChanged，以确保当TextBox包含文本时，Watch!按钮是启用的。

```
private void LocationBox_TextChanged(object sender, TextC
```

```
{
```



```

        WatchButton.IsEnabled = !string.IsNullOrEmpty(LocationBox.Text);
    }
}

```

（9）将以下代码添加到Watch!按钮的Click事件处理程序，这会启动FileSystemWatcher:

```

private void WatchButton_Click(object sender, RoutedEventArgs e)
{
    // Create a new FileSystemWatcher and set its properties.
    watcher = new FileSystemWatcher();
    watcher.Path = System.IO.Path.GetDirectoryName(LocationBox.Text);
    watcher.Filter = System.IO.Path.GetFileName(LocationBox.Text);
    watcher.NotifyFilter = NotifyFilters.Attributes |
        NotifyFilters.CreationTime |
        NotifyFilters.FileName |
        NotifyFilters.LastWrite |
        NotifyFilters.Size;
    watcher.Changed += Watcher_Changed;
    watcher.Created += Watcher_Changed;
    watcher.Deleted += Watcher_Changed;
    watcher.Renamed += Watcher_Changed;
    watcher.EnableRaisingEvents = true;
}

```

```
        watcher.NotifyFilter = NotifyFilters.LastWrite |  
  
        NotifyFilters.FileName|NotifyFilters.Size;  
  
        AddMessage("Watching " + LocationBox.Text);  
  
        // Begin watching.  
  
        watcher.EnableRaisingEvents = true;  
  
    }
```

(10) 创建目录C:\TempWatch, 在该目录中创建文件temp.txt。

(11) 运行应用程序。如果成功地构建了所有内容，则单击**Browse**按钮，并选择C:\TempWatch\temp.txt。

(12) 单击**Watch!**按钮，开始监控文件。在应用程序中可以见到的唯一变化是有一条消息，指出正在监控文件。

(13) 使用Windows资源管理器导航到C:\TempWatch。在记事本中打开temp.txt，并在文件中添加一些文本。保存此文件。

(14) 重命名该文件。

(15) 可以看到对所监控文件的变动说明，如图18-8所示。

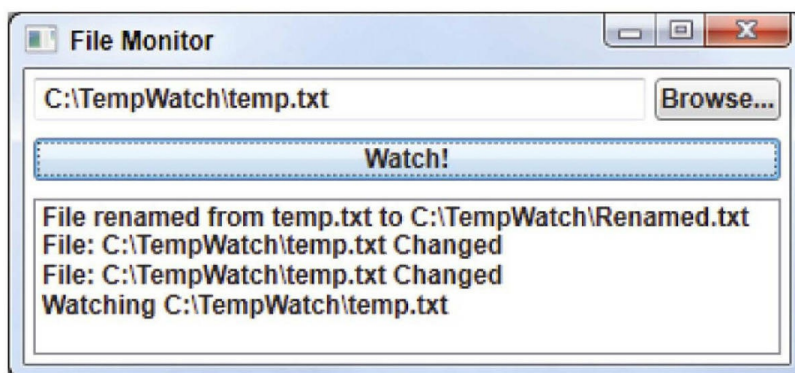


图18-8

示例的说明

此应用程序非常简单，但它演示了FileSystemWatcher的工作原理。尝试在监控文本框中输入不同字符串。如果在目录中指定*.*，程序就会监控目录中的所有变化。

应用程序中的大多数代码都用于建立FileSystemWatcher对象，以监控正确的位置：

```

watcher.Path = System.IO.Path.GetDirectoryName(LocationE
watcher.Filter = System.IO.Path.GetFileName(LocationBox.
watcher.NotifyFilter = NotifyFilters.LastWrite |
    NotifyFilters.FileName|NotifyFilters.Size;
AddMessage("Watching " + LocationBox.Text);
// Begin watching.
watcher.EnableRaisingEvents = true;

```

代码首先设置要监控的目录的路径。这使用了到现在尚未介绍的一个新对象System.IO.Path。这是一个静态类，非常类似于静态File对象。它给出了许多静态方法，以处理和提取文件位置字符串中的信息。这里首先使用它通过GetDirectoryName()方法提取用户在文本框中输入的目录名。

下一行代码设置对象的过滤器，过滤器可以是一个实际文件，表示仅监控该文件。过滤器也可以是*.txt，表示要监控指定目录中的所有.txt文件。我们也可以使用Path静态对象从所提供的文件位置中提取信息。

NotifyFilter是NotifyFilters枚举值的组合，指定组成变化的内容。在此，如果最后写入的时间信息、文件名称或文件大小发生了变化，它就将此变化通知应用程序。更新UI后，将EnableRaisingEvents属性设置为true，开始监控。

但在此之前，还要创建对象，设置事件处理程序。

```

watcher = new FileSystemWatcher();
watcher.Deleted += (s, e) =>
    AddMessage($"File: {e.FullPath} Deleted");

```

```
watcher.Renamed += (s, e) =>
    AddMessage($"File renamed from {e.OldName} to {e.FullP
watcher.Changed += (s, e) =>
    AddMessage($"File: {e.FullPath} {e.ChangeType.ToString
watcher.Created += (s, e) =>
    AddMessage($"File: {e.FullPath} Created");
```

这段代码使用了Lambda表达式来创建匿名事件处理程序，当删除、重命名、修改或创建文件时，监控器对象就触发事件，调用这些事件处理程序。这些处理程序简单地调用AddMessage()方法来添加一条消息。显然，根据应用程序的不同，还可以有更复杂的响应。在目录中添加文件时，可将其移到别处，或读取其内容，引发新的进程。其可能的用法是无穷无尽的！

18.4 练习

- (1) 只有导入哪个名称空间才允许应用程序使用文件？
- (2) 何时使用`FileStream`对象，而不是使用`StreamWriter`对象写入文件？
- (3) `StreamReader`类的哪些方法允许从文件中读取数据，每个方法的具体作用是什么？
- (4) 哪个类可使用`Deflate`算法来压缩流？
- (5) `FileSystemWatcher`类提供了哪些事件，其作用是什么？
- (6) 修改本章构建的`FileWatch`应用程序，使得不必退出应用程序就可以打开和关闭文件系统监控功能。

附录A给出了练习答案。

18.5 本章要点

主题	要点
流	流是序列化设备的一种抽象表示，可以一次从序列化设备中读取或写入一个字节。文件就是这种设备的一个例子。流有两种类型：输入和输出，分别用于读取和写入设备
文件访问类	.NET Framework中具有大量抽象了文件系统访问的类，包括通过静态方法处理文件和目录的File和Directory，可实例化为表示特定文件和目录的FileInfo和DirectoryInfo。后两个类在对文件和目录执行多个操作时使用，因为这两个类不要求为每个方法调用指定路径。可以在文件和目录上执行的典型操作包括查看和修改属性，以及创建、删除和复制操作
文件路径	文件和目录路径可以是绝对的或相对的。绝对路径给出了某位置的完整描述，从包含它的驱动器的根目录开始。所有父目录都与子目录用反斜杠隔开。相对路径与之类似，但从文件系统的指定点开始，例如执行应用程序的目录（工作目录）。浏览文件系统时，常使用父目录的别名..
FileStream对象	FileStream对象允许访问文件的内容，以进行读写。它以字节为单位访问文件数据，所以并不总是访问文件数据的最佳选项。FileStream实例维护着文件内部的一个位置字节索引，这样就可以浏览文件的内容了。以这种方式访问文件的任意位置称为随机访问
	一种读写文件数据的更简便方法是使用StreamReader和StreamWriter类，以及FileStream。它们允许读写字符和字符串数据，而不是处理字节。这些类型提供了我们熟

读写流	悉的处理字符串的方法，包括ReadLine()和WriteLine()。因为它们处理的是字符串数据，所以使用这些类，可以更加方便地处理逗号分隔的文件（这是表示结构化数据的常见方式）
压缩文件	可使用DeflateStream和GZipStream压缩流类来读写文件中的压缩数据。这些类与FileStream一样，也处理字节数据，但可以通过StreamReader和StreamWriter类访问数据，以简化代码
监控文件系统	可使用FileSystemWatcher类监控文件系统数据的变化。可以监控文件和目录，如有必要，还可以提供一个过滤器，根据需要仅修改有特定扩展名的文件。 FileSystemWatcher实例通过触发事件，来通知我们发生了变化，这些事件可以在代码中处理

第19章 XML和JSON

本章内容：

- XML基础
- JSON基础
- XML模式
- XML文档对象模型
- 把XML转换为JSON
- 使用XPath搜索XML文档

本章源代码下载：

本章的WROX.COM代码下载在
www.wrox.com/go/beginningvisualc#2015programming的Download Code
选项卡上。代码在Chapter 19 download中，根据本章的名称单独命名
了。

C#编程语言以机器和人类均可读的格式描述了计算机逻辑，而XML和JSON都是数据语言，以简单的文本格式存储数据，这意味着它可以被人类和几乎任何计算机读取。

大多数C#.NET应用程序都使用XML以某种形式来存储数据，如.config文件用于存储配置细节，XAML文件在WPF和Windows Store应

用程序中使用。因为这个重要的事实，本章将花最多的时间介绍XML，只简要介绍JSON。

本章将学习XML和JSON的基础知识，然后学习如何创建XML文档和模式。你将学习XmlDocument类的基本知识，如何读写XML，如何插入和删除节点，如何将XML转换成JSON格式，最后学习如何在XML文档中使用XPath搜索数据。

19.1 XML基础

可扩展标记语言（XML）是一种数据语言，它将数据存储在简单的文本格式里，可以被人类和几乎任何计算机理解。它是一种W3C标准格式，类似于HTML（www.w3.org/XML）。Microsoft在.NET Framework和其他微软产品中已经完全采用它。即使是Microsoft Office的新版本引入的文档格式也是基于XML的，但Office应用程序本身不是.NET应用程序。

XML的细节非常复杂，因此在此不介绍其所有细节。幸好，大多数任务都不需要了解XML的详细知识，因为Visual Studio通常会处理其中大多数工作——我们基本上不必手动编写XML文档。如果想更深入地了解XML，可以阅读Joe Fawcett、Danny Ayers和Liam Quin（Wrox, 2012）编写的*Beginning XML*，或许多在线教程，比如www.xmlnews.org/docs/xml-basics.html或<http://www.w3schools.com/xml/>。

XML的基本格式很简单，下面的例子显示了共享图书数据的XML格式。

```
<book>
  <title>Beginning Visual C# 2015</title>
  <author>Benjamin Perkins et al</author>
  <code>096689</code>
</book>
```

在这个例子中，每本书都有书名、作者和标识这本书的独特代码。每本书的数据包含在一个book元素中，该元素用<book>开始标记开头，用</book>结束标记结束。标题、作者和代码值存储在book元素的嵌套元素中。

元素的标签内也可能有特性。如果书的代码是book元素的一个特性，而不是一个元素，book元素的开头就是<book code=096689>。为简单起见，本章的例子仅使用元素。一般，特性和元素都称为节点，类似于图中的节点。

19.2 JSON基础

开发C#应用程序时，另一门可能遇到的数据语言是JSON。JSON表示JavaScript Object Notation。就像XML一样，它也是一个标准（www.json.org），尽管从名字上来看，它来源于JavaScript语言而非C#。虽然JSON不像XML一样在整个.NET中使用，但它是传输Web服务和Web浏览器中数据的一种常见格式。

JSON也有一个非常简单的格式。此前用XML显示的图书数据在JSON中显示为：

```
{"book":[{"title":"Beginning Visual C# 2015",  
          "author":"Benjamin Perkins et al",  
          "code":"096689"}]}
```

与之前的XML的示例一样，这里也显示了书名、作者和唯一代码。JSON使用花括号（{ }）分隔数据块，使用方括号（[]）界定数组，其方式与C#、JavaScript和其他C语言相似，它们也给代码块使用花括号，给数组使用方括号。

JSON是一种比XML更紧凑的格式，但是人们很难阅读它，特别是复杂数据会使用很多花括号和括号进行深度嵌套。

19.3 XML模式

XML文档可以用模式类描述，模式是另一个XML文件，描述了允许在一个特定的文档中使用的元素和特性。可以根据模式验证XML文档，确保程序不会遇到不打算处理的数据。用于C#的标准模式XML格式是XSD（XML Schema Definition）。

图19-1包含了VS能够识别的模式的一个长列表，但它不会自动记忆已使用过的模式。如果经常使用某个模式，但不想在每次需要时浏览找到该模式，就可以把该模式复制到以下位置：C:\Program Files\Microsoft Visual Studio 14.0\Xml\Schemas。复制到该位置的任何模式都会显示在XML Schemas对话框中。

试一试：在VS中创建XML文档：Chapter19\XML and Schema\GhostStories.xml

按照下面的步骤来创建XML文档。

（1）打开VS，从菜单中选择File|New|File。如果没有看到这个选项，请创建一个新项目。在Solution Explorer中右击项目，选择添加一个新项。然后从对话框中选择XML File。

（2）在New File对话框中，选择XML File，单击Open。VS会自动创建一个新的XML文档。如图19-1所示，VS添加了XML声明，以

encoding特性结束（其特性和元素也是彩色的，但是在黑白纸张中看不到这种效果）。

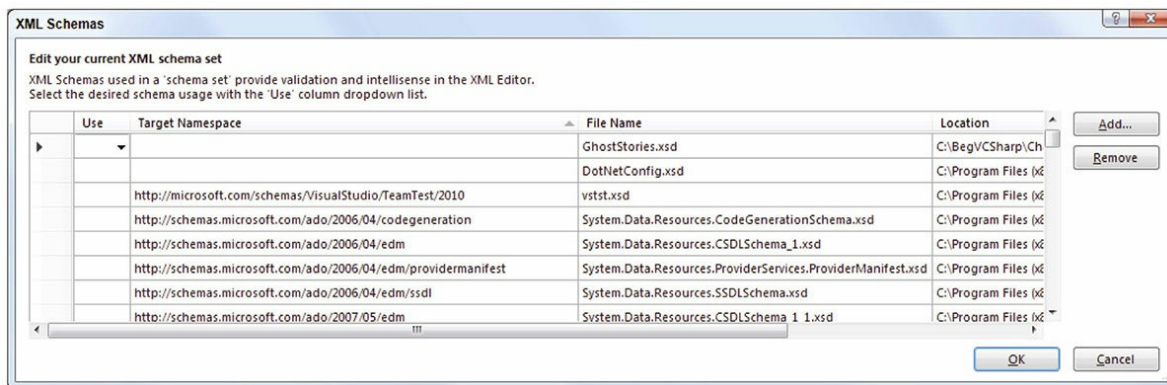


图19-1

（3）按下Ctrl+S组合键，或者在菜单中选择File|Save XMLFile1.xml，保存文件。VS会询问文件的保存位置，以及文件的名称。将其保存在BeginVCSharp\Chapter19\XML and Schemas文件夹中，取名为GhostStories.xml。

（4）将光标移到XML声明下面的代码行，输入文本<stories>。注意，输入大于号关闭开始标记时，VS会自动置入结束标记。

（5）输入下面的XML文件，单击Save:

```
<stories>
  <story>
    <title>A House in Aungier Street</title>
    <author>
      <name>Sheridan Le Fanu</name>
      <nationality>Irish</nationality>
```

```
        </author>
    <rating>eerie</rating>
</story>
<story>
    <title>The Signalman</title>
    <author>
        <name>Charles Dickens</name>
        <nationality>English</nationality>
    </author>
    <rating>atmospheric</rating>
</story>
<story>
    <title>The Turn of the Screw</title>
    <author>
        <name>Henry James</name>
        <nationality>American</nationality>
    </author>
    <rating>a bit dull</rating>
</story>
</stories>
```

(6) 现在可以让Visual Studio为刚才编写的XML文件创建相应的模式。为此，从XML菜单中选择Create Schema菜单项，单击Save as GhostStories.xsd，保存得到的XSD文件。

(7) 返回到XML文件，在结束标记</stories>之前输入如下XML：

```
<story>
```



```
<title>Number 13</title>
  <author><name>M.R. James</name>
    <nationality>English</nationality>
  </author>
<rating>mysterious</rating>
</story>
```

注意，开始输入开始标记时，会显示IntelliSense提示。这是因为Visual Studio知道把新建的XSD模式连接到正在输入的XML文件上。

（8）可在Visual Studio中创建XML和一个或多个模式之间的链接。选择XML|Schemas，会打开如图19-2所示的对话框。在Visual Studio可识别的长模式列表顶部，会看到GhostStories.xsd。在它的左边是一个复选标记，表示这个模式用于当前的XML文档。

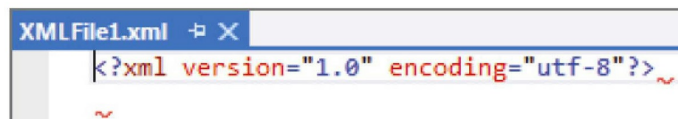


图19-2

19.4 XML文档对象模型

XML文档对象模型（Document Object Model，DOM）是一组以非常直观的方式访问和处理XML的类。DOM不是读取XML数据的最快捷方式，但只要理解了类和XML文档中元素之间的关系，DOM就很容易使用。

构成DOM的类在名称空间System.Xml中。在这个名称空间中有几个类和子名称空间。但本章只介绍几个便于操作XML的类。要学习和使用的类如表19-1所示。

表19-1 常用的DOM类

类名	说明
XmlNode	这个类表示文档树中的一个节点，它是本章许多类的基类。如果这个节点表示XML文档的根，就可以从它导航到文档的任意位置
XmlDocument	扩展了XmlNode类，但通常是使用XML的第一个对象，因为这个类用于加载磁盘或其他地方的数据并在这些位置保存数据
XmlElement	表示XML文档中的一个元素。XmlElement派生于XmlNode，XmlNode派生于XmlNode
XmlAttribute	表示一个特性，与XmlAttribute类一样，它也派生于XmlNode类
XmlText	表示开始标记和结束标记之间的文本

XmlComment	表示一种特殊类型的节点，这种节点不是文档的一部分，但为阅读器提供文档各部分的信息
XmlNodeList	表示一个节点集合

19.4.1 XmlDocument类

通常，要处理XML的应用程序，首先应从磁盘中读取它。如表19-1所示，这是XmlDocument类的工作。可以将XmlDocument看成磁盘上文件的内存表示。使用XmlDocument类把文件加载到内存后，就可以从中获得文档的根节点，开始读取和处理XML了：

```
using System.Xml;

.
.
.

XmlDocument document = new XmlDocument();
document.Load(@"C:\BegVCSharp\Chapter19\XML and Schema\books.x
```

这两行代码创建了XmlDocument类的一个新实例，并在其中加载books.xml文件。

注意：文件夹名是一个绝对路径，文件夹结构可以不同，此时，应调整document.Load中的路径，以反映计算机中实际的文件夹路径。

Xml Document类位于System.Xml名称空间中，所以应在代码开头的using部分插入using System.Xml;语句。

除了加载和保存XML外，XmlDocument类还负责维护XML结构。所以，这个类有许多方法可以用于创建、修改和删除树中的节点。稍后将介绍其中一些方法，但为了正确理解这些方法，还需要了解另一个类：XmlElement。

19.4.2 XmlDocument类

文档加载到内存后，就要对它执行一些操作。上面代码创建的XmlDocument实例的DocumentElement属性会返回一个XmlElement实例（表示XmlDocument的根节点）。这个元素非常重要，因为有了它，就可以访问文档中的所有信息。

```
XmlDocument document = new XmlDocument();  
document.Load(@"C:\BegVCSharp\Chapter19\  
XML and Schema\books.xml");  
XmlElement element = document.DocumentElement;
```

获得文档的根节点后，就可以使用信息了。XmlElement类包含的方法和属性可以处理树的节点和特性。下面首先看看用于导航XML元素的属性，如表19-2所示。

表19-2 XmlElement的属性

属性	说明
FirstChild	<p>该属性返回当前节点之后的第一个子节点。在本章前面的books.xml文件中，文档的根节点是books，根节点之后的节点是book，在该文档中，根节点books的第一个子节点是book。</p> <pre> <books> Root node <book> FirstChild </pre> <p>FirstChild返回一个XmlNode对象，应测试返回节点的类型，因为它不总是一个XmlElement实例。在books示例中，Title元素的子元素是表示文本Beginning Visual C#的XmlText节点</p>
LastChild	<p>该属性的操作与FirstChild属性十分类似，但返回当前节点的最后一个子节点。在books示例中，books节点的最后一个子节点仍是book，但它表示"Beginning XML" book。</p> <pre> <books> Root node <book> FirstChild <title>Beginning Visual C# 2015</title> <author>Benjamin Perkins et al</author> <code>096689</code> </book> <book> LastChild <title>Beginning XML</title> <author>Joe Fawcett at al</author> <code>162132</code> </book> </books> </pre>

ParentNode	该属性返回当前节点的父节点。在books示例中，books节点是book节点的父节点
NextSibling	FirstChild和LastChild属性返回当前节点的叶子节点，而NextSibling节点返回有相同父节点的下一个节点。在books示例中，title元素的NextSibling属性返回author元素，在author元素上调用NextSibling，会返回code元素
HasChildNodes	检查当前元素是否有子元素，而不必获取FirstChild的值并检查它是否为null

使用表19-2中的5个属性，可以遍历整个XmlDocument，如下面的示例所示。

试一试：迭代XML文档中的所有节点：

Chapter19\LoopThroughXmlDocument\MainWindows.xaml.

在这个示例中，要创建一个小型WPF应用程序，迭代XML文档中的所有节点，打印出元素的名称，如果是XmlText元素，就打印出包含在元素中的文本。这段代码使用了Books.xml，如前面的“模式”一节所述。如果没有创建这个文件，可在本书的下载代码中找到它（Chapter19\XML and Schemas\）。

（1）首先创建一个新的WPF项目。选择File|New|Project菜单项，在打开的对话框中选择Windows|WPF Application。将项目命名为LoopThroughXmlDocument，按下回车键。

（2）将一个TextBlock和一个Button控件拖放到窗体上，按照图19-

3所示设计窗体。

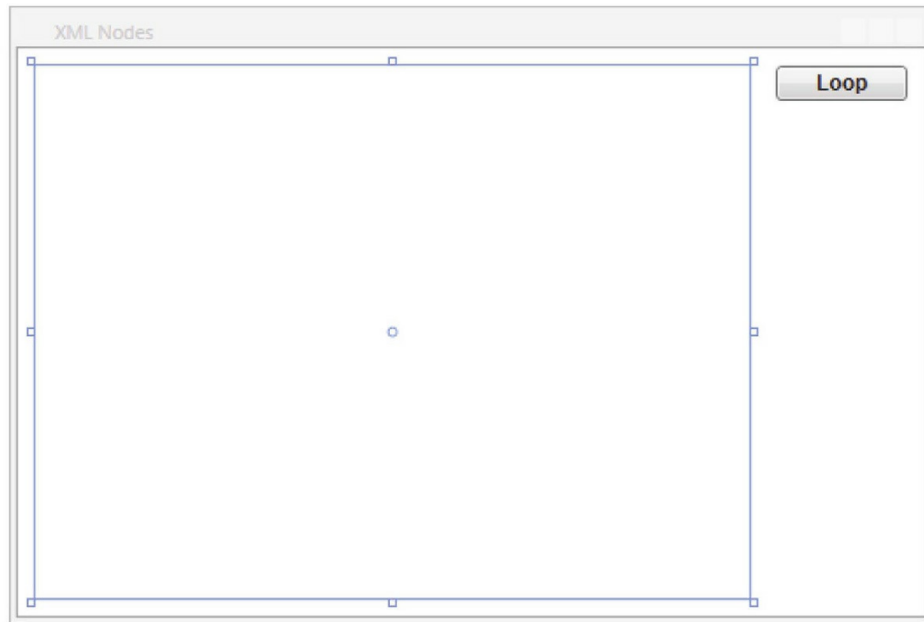


图19-3

(3) 将TextBlock控件命名为textBlockResults，按钮命名为buttonLoop。允许TextBlock填满按钮没有使用的全部空间。

(4) 为按钮的单击事件添加事件处理程序，输入下面的代码。注意，要在文件顶部的using部分添加using System.Xml;:

```
private void buttonLoop_Click(object sender, RoutedEventArgs e)
{
    XmlDocument document = new XmlDocument();
    document.Load(booksFile);
    textBlockResults.Text =
        FormatText(document.DocumentElement as XmlNode, "", "")
}
```

```

private string FormatText(XmlNode node, string text, string
{
    if (node is XmlText)
    {
        text += node.Value;
        return text;
    }
    if (string.IsNullOrEmpty(indent))
        indent = "";
    else
    {
        text += "\r\n" + indent;
    }
    if (node is XmlComment)
    {
        text += node.OuterXml;
        return text;
    }
    text += "<" + node.Name;
    if (node.Attributes.Count > 0)
    {
        AddAttributes(node, ref text);
    }
    if (node.HasChildNodes)
    {
        text += ">";
        foreach (XmlNode child in node.ChildNodes)

```



```

    {
        text = FormatText(child, text, indent + " ");
    }
    if (node.ChildNodes.Count == 1 &&
        (node.FirstChild is XmlText || node.FirstChild is XmlElement))
        text += "</" + node.Name + ">";
    else
        text += "\r\n" + indent + "</" + node.Name + ">";
    }
    else
        text += " />";
    return text;
}

private void AddAttributes(XmlNode node, ref string text)
{
    foreach (XmlAttribute xa in node.Attributes)
    {
        text += " " + xa.Name + "='" + xa.Value + "'";
    }
}
}

```

(5) 添加一个私有常量，用于保存所加载文件的位置。需要把这个常量的值改为本地系统中存储文件的位置：

```

private const string booksFile =
    @"C:\BegVCSharp\Chapter19\XML and Schema\Books.xml";

```

(6) 运行该应用程序，单击Loop按钮。结果如图19-4所示。

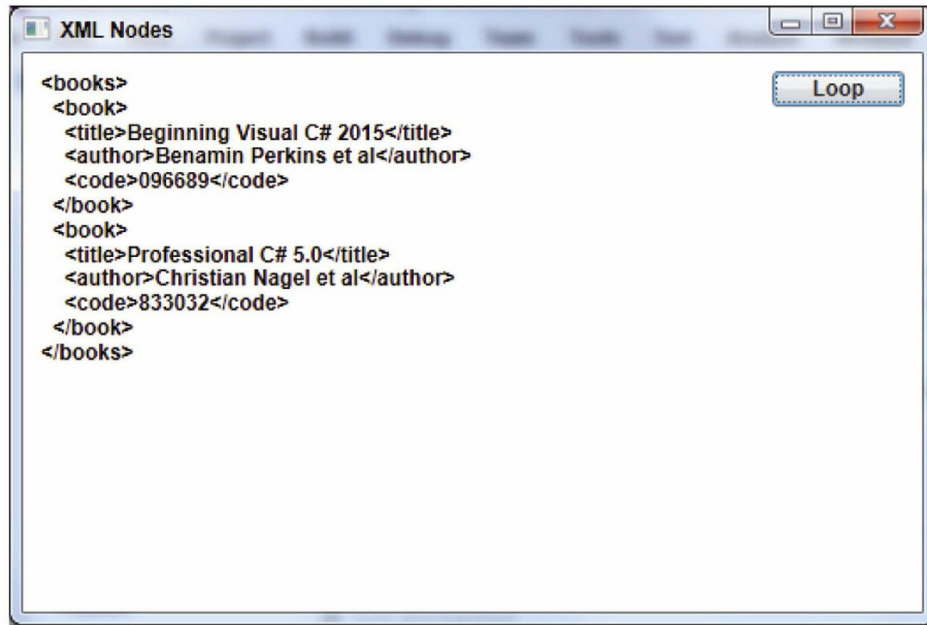


图19-4

示例的说明

单击按钮时，会调用XmlDocument方法Load。这个方法把文件中的XML加载到XmlDocument实例中，XmlDocument实例用于访问XML的元素。接着调用一个方法来递归迭代XML，并把XML文档的根节点传送给方法。根元素是使用XmlDocument类的属性DocumentElement获得的。除了在传送给FormatText方法的根参数上检查null外，还要注意if语句：

```
if (node is XmlText)
{
    ...
}
```

is运算符可以检查对象的类型，如果实例是指定的类型，就返回

true。即使根节点声明为XmlNode，这也只是要操作的对象的基本类型。使用is运算符可以在运行期间确定对象类型，并根据该类型选择要执行的操作。

在FormatText方法中给文本框生成文本。注意必须知道根节点的当前实例的类型，因为要显示的信息的获取方式对于不同的元素来说是不同的。我们要显示XmlElement元素的名称和XmlText元素的值。

19.4.3 修改节点的值

在了解如何改变节点值之前，先要明白，节点值一般比较复杂。实际上，即使派生于XmlNode的所有类都包含Value属性，它也很少返回有用的信息。初看起来它可能令人失望，但实际上是十分合理的。分析一下前面的books示例：

```
<books>
  <book>
    <title>Beginning Visual C# 2015</title>
    <author>Benjamin Perkins et al</author>
    <code>096689</code>
  </book>
</book>
</books>
```

文档中的每对标记都解析为DOM中的一个节点。在迭代文档中的所有节点时，会遇到许多XmlElement节点和三个XmlText节点。上述

XML中的XmlElement节点是<books>、<book>、<title>、<author>和<code>。XmlText节点是title、author和code开始标记和结束标记之间的文本。也可以说title、author和code的值是标记之间的文本，但文本本身就是一个节点，是这个节点实际包含了值。其他标记都没有相关的值。

在上述FormatText方法的代码靠近顶部的位置，if块中的下述代码在当前节点是XmlText时执行：

```
text += node.Value;
```

XmlText节点实例的Value属性用于获取节点的值。

如果使用XmlElement类型的节点的Value属性，就返回null，但如果使用另两个方法InnerText和InnerXml中的一个，就可以获取XmlElement开始标记和结束标记之间的信息。也就是说，可以使用两个方法和一个属性来操作节点的值，如表19-3所示。

表19-3 获取节点值的三种方法

属性	说明
InnerText	这个属性获取当前节点中所有子节点的文本，把它作为一个串联字符串返回。也就是说，在上面的XML中，如果获取book节点的InnerText值，就返回字符串Beginning Visual C 2015#Benjamin Perkins et al096689。如果获取title节点的InnerText，就只返回Beginnning Visual C# 2015。可以使用这个方法设置文本，但要小心，因为如果设置了错误节点的文本，就很可能改写不想改变的信息
	InnerXml属性返回类似于InnerText的文本，也返回所有标记。如果获取book节点上的InnerXml值，结果是如下字符

InnerXml	<p>串：</p> <pre><title>Beginning Visual C# 2015</title> <author>Benjamin Perkins et al </author><code>096689</code></pre> <p>可以看出，如果字符串包含要直接插入XML文档的内容，这是很有用的。但是要对该字符串负全责，如果插入格式错误的XML，应用程序就会生成异常</p>
Value	<p>Value属性是操作文档中信息的最精练方式，但如前所述，在获取值时，只有几个类会返回有用的信息。返回所需文本的类如下所示：</p> <pre>XmlText XmlComment XmlAttribute</pre>

1. 插入新节点

了解了如何遍历XML文档，如何获取元素的值后，下面学习如何给前面使用的books文档添加节点，改变文档的结构。

要在列表中插入新元素，需要使用XmlDocument和XmlNode类中的新方法，如表19-4所示。可使用XmlDocument类的方法创建新的XmlNode和XmlElement实例，这非常不错，因为这两个类都只有一个受保护的构造函数，不能直接使用new创建它们的实例。

表19-4 用于创建节点的方法

方法	说明

CreateNode	创建任意类型的节点。该方法有三个重载版本，其中两个允许创建XmlNodeType枚举中所列出的类型的节点，另一个允许把要使用的节点类型指定为字符串。除非对指定的不是枚举中的节点类型有完全的把握，否则强烈推荐使用枚举的两个重载版本。该方法返回一个XmlNode实例，该实例可以显式地转换为合适的类型
CreateElement	这只是CreateNode的一个版本，只能创建XmlElement类型的节点
CreateAttribute	这也只是CreateNode的一个版本，只能创建XmlAttribute类型的节点
CreateTextNode	创建XmlTextNode类型的节点
CreateComment	在这个列表中包含这个方法，是为了说明可以创建的节点类型的多样性。该方法并不创建由XML文档表示的数据节点，而是创建注释，以便人们读取数据。在应用程序中读取文档时，就可以读取注释

表19-4中的方法都用于创建节点，在调用其中一个方法后，就必须执行一些操作。在创建节点后，节点并没有包含其他信息，节点也没有插入到文档中。为此，应使用派生于XmlNode的类（包括XmlDocument和XmlElement）中的方法。表19-5描述了这些方法。

表19-5 用于插入节点的方法

方法	说明
AppendChild	把一个子节点追加到XmlNode类型或其派生类型的节点上。在调用该方法后，追加的节点显示在相应节点的子节点列表的最后。如果不关心子节点的顺序，这就不重要，但如果子节点的顺序很重要，就应按正确

	的顺序追加节点
InsertAfter	使用InsertAfter方法，可以控制插入新节点的位置。该方法带有两个参数，第一个是新节点，第二个是在其后插入新节点的节点
InsertBefore	这个方法与InsertAfter类似，但新节点插到参考节点之前

下面的示例以前面的示例为基础，在books.xml文档中插入一个book节点。该示例中（目前）没有清理文档的代码，所以如果该示例运行几次，文档中就会包含许多相同的节点。

试一试：创建节点：

Chapter19\LoopThroughXmlDocument\MainWindow.xaml.c

本例以前面创建的LoopThroughXmlDocument项目为基础。按照下面的步骤给books.xml文档添加一个节点。

（1）将TextBlock放在一个ScrollViewer中，将其VerticalScrollBarVisibility属性设为Auto。

（2）在窗体的已有按钮下面添加一个按钮，命名为buttonCreateNode。将其Content属性改为Create。

（3）为新按钮添加单击事件的处理程序，输入下面的代码：

```
private void buttonCreateNode_Click(object sender, RoutedEventArgs)
{
```

```

// Load the XML document.
XmlDocument document = new XmlDocument();
document.Load(booksFile);
// Get the root element.
XmlElement root = document.DocumentElement;
// Create the new nodes.
XmlElement newBook = document.CreateElement("book");
XmlElement newTitle = document.CreateElement("title");
XmlElement newAuthor = document.CreateElement("author");
XmlElement newCode = document.CreateElement("code");
XmlText title = document.CreateTextNode("Beginning Visual
XmlText author = document.CreateTextNode("Karli Watson et
XmlText code = document.CreateTextNode("314418");
XmlComment comment = document.CreateComment("The previous
// Insert the elements.
newBook.AppendChild(comment);
newBook.AppendChild(newTitle);
newBook.AppendChild(newAuthor);
newBook.AppendChild(newCode);
newTitle.AppendChild(title);
newAuthor.AppendChild(author);
newCode.AppendChild(code);
root.InsertAfter(newBook, root.LastChild);
document.Save(booksFile);
}

```

(4) 运行应用程序，单击Create按钮。再单击Loop按钮，对话框

如图19-5所示。

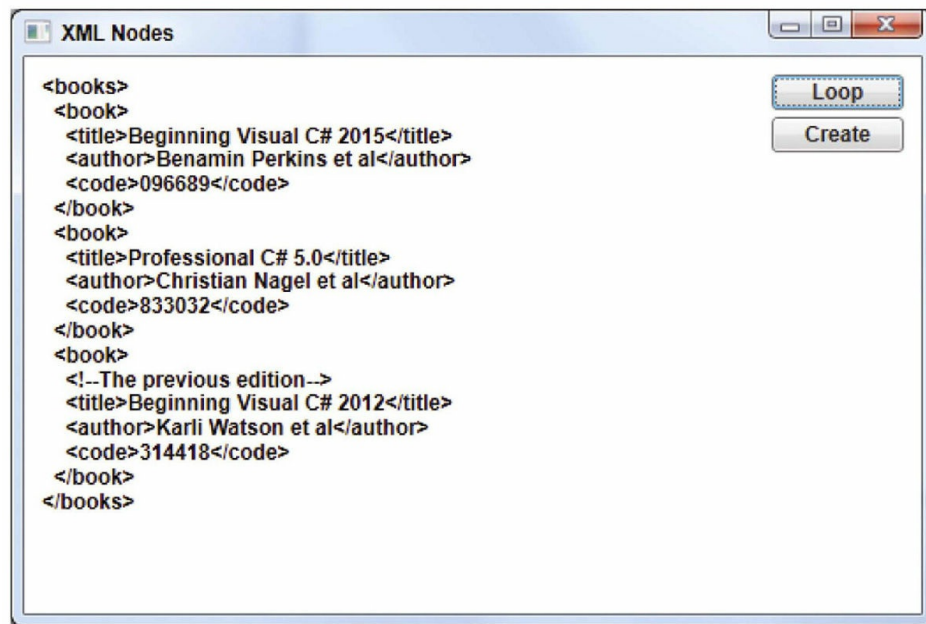


图19-5

在上面的示例中没有创建一个重要类型的节点：XmlAttribute，这里把它留作本章的练习。

示例的说明

buttonCreateNode_Click方法中的代码创建了所有节点。它创建了8个新节点，其中4个节点的类型是XmlElement，3个节点的类型是XmlText，最后一个节点的类型是XmlComment。

所有节点都是使用封装XmlDocument实例的方法创建的。XmlElement节点是用CreateElement方法创建的，XmlText节点是用CreateTextNode方法创建的，XmlComment节点是用CreateComment方法创建的。

创建完节点后，还需要把它们插入XML树。这是使用元素上的AppendChild方法实现的，新节点将成为该元素的一个子节点。唯一的例外是book节点，它是所有新节点的根节点。这个节点使用根对象的InsertAfter方法插入到树中。使用AppendChild方法插入的所有节点总是成为子节点列表的最后一项，而InsertAfter允许指定节点的位置。

2. 删除节点

学习了如何创建新节点，剩下的就是如何删除节点了。派生于XmlNode的所有类都包含允许从文档中删除节点的两个方法，如表19-6所示。

表19-6 用于删除节点的方法

方法	说明
RemoveAll	这个方法删除节点上的所有子节点。不太明显的是，它还会删除节点上的所有特性，因为它们也被看成子节点
RemoveChild	这个方法删除节点上的一个子节点，返回从文档中删除的节点。如果改变主意，还可以把它重新插回到文档

下面的示例将扩展前面两个示例创建的应用程序，使其包含删除节点的功能。只需要找出book节点的最后一个实例，并删除它。

试一试：删除节点：

Chapter19\LoopThroughXmlDocument\MainWindow.xaml.c

本例以前面创建的LoopThroughXmlDocument项目为基础。下面的步骤将找出book节点的最后一个实例，并删除它。

(1) 在前面两个已有的按钮下面添加一个新按钮，命名为buttonDeleteNode，将其Content属性设置为Delete。

(2) 双击新按钮，输入下面的代码：

```
private void buttonDeleteNode_Click(object sender, RoutedEventArgs)
{
    // Load the XML document.
    XmlDocument document = new XmlDocument();
    document.Load(booksFile);
    // Get the root element.
    XmlElement root = document.DocumentElement;
    // Find the node. root is the<books> tag, so its last child
    // which will be the last<book> node.
    if (root.HasChildNodes)
    {
        XmlNode book = root.LastChild;
        // Delete the child.
        root.RemoveChild(book);
        // Save the document back to disk.
```

```
        document.Save(booksFile);  
    }  
}
```

(3) 运行应用程序。单击Delete Node按钮，再单击Loop按钮，树中的最后一个节点就会消失。

示例的说明

把XML加载到XmlDocument对象上后，就检查根元素，确定在加载的XML中是否有子元素。如果有，就使用XmlElement类的LastChild属性获取最后一个子元素。之后，调用RemoveChild，给它传送要删除的元素实例（在本例中是根元素的最后一个子元素），就删除了该元素。

3. 选择节点

前面介绍了如何浏览XML文档、如何操作文档的值、如何创建新节点以及如何删除它们。剩下的就是在不遍历整个树的情况下选择节点。

XmlNode类包含的两个方法常用于从文档中选择节点，且不遍历其中的每个节点，如表19-7所示。这两个方法是SelectSingleNode和SelectNodes，它们都使用一种特殊的查询语言XPath来选择节点。稍后将介绍该语言。

表19-7 用于选择节点的方法

方法	说明
SelectSingleNode	选择一个节点。如果创建一个查找多个节点的查询，就只返回第一个节点
SelectNodes	以XmlNodeList类的形式返回一个节点集合

19.5 把XML转换为JSON

本章开头提到JSON数据语言。C#系统库有限地支持JSON，但是可以使用一个免费的第三方JSON库，将XML转换为JSON，将JSON转换为XML，用JSON完成其他操作，类似于用.NET类处理XML。在Visual Studio中，可以通过NuGet Package Manager获得的一个这样的库是Newtonsoft JSON.NET包。这个包的帮助和完整教程可在www.json.net上获得。

下面的简短例子扩展了本章前面示例创建的应用程序，能将XML转换为JSON。

试一试：转换：

Chapter19\LoopThroughXmlDocument\MainWindow.xaml.c

这个例子是建立在前面创建的LoopThroughXmlDocument项目的基础上。以下步骤可以找到、删除book节点的最后一个实例：

(1) 在Visual Studio菜单中，进入Tools|NuGet Package Manager|Manage NuGet Packages for Solution。选择Newtonsoft.Json包，如图19-6所示。单击Install按钮，然后在Review Changes对话框中单击OK，完成安装。

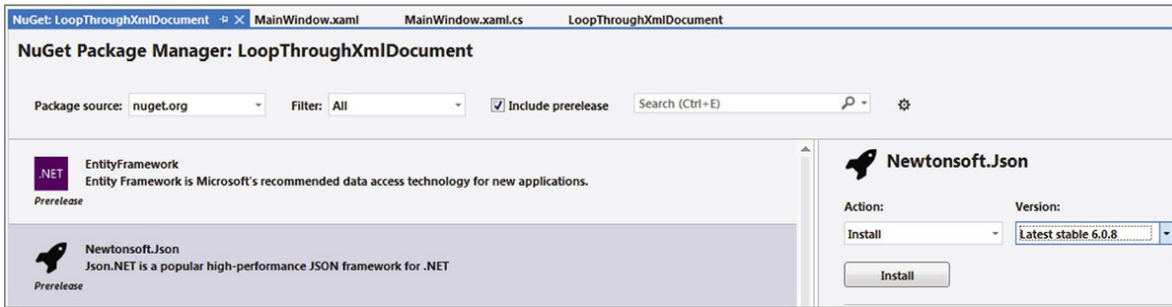


图19-6

(2) 在三个已有按钮的下面添加一个新按钮，并命名为buttonXMLtoJSON。把它的Content属性设置为XML>JSON。

(3) 双击新按钮，输入以下代码：

```
private void buttonXMLtoJSON_Click(object sender, RoutedEventArgs)
{
    // Load the XML document.
    XmlDocument document = new XmlDocument();
    document.Load(booksFile);
    string json = Newtonsoft.Json.JsonConvert.SerializeXmlNode(
        textBlockResults.Text = json;
}
```

(4) 运行应用程序。单击XML>JSON按钮。本书的JSON版本数据就显示在主窗口中，如图19-7所示。

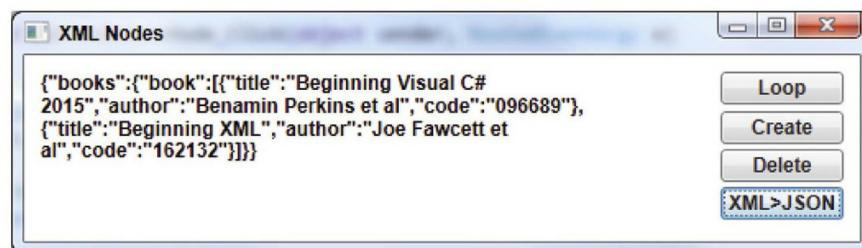


图19-7

示例的说明

开头几步把XML加载到XmlDocument对象，此后调用Newtonsoft JSON包方法JsonConvert.SerializeXmlNode将XML文档转换为JSON格式的文本字符串。接着在已加载的XML的任何子元素中显示JSON文本。如果有任何子元素，就使用textBlockResults窗口。可以看出，JSON版本的图书数据比XML更紧凑，但有点难以阅读。所以，JSON常用于网络上的数据传输，而不是存储在可能由人们直接读取的文件里。

19.6 用XPath搜索XML

XPath是XML文档的查询语言，就像SQL是关系数据库的查询语言一样。它由表19-7中的两个方法使用，以免遍历XML文档的整个树。但是需要花一定的时间才能熟悉它，因为其语法与SQL或C#完全不同。

注意： XPath相当复杂，这里只介绍其中的一小部分，就足以选择节点了。如果想了解XPath的更多内容，可参阅 www.w3.org/TR/xpath和Visual Studio帮助页面。

为正确使用XPath，下面要使用XML文件Elements.xml。该文档包含元素周期表中的部分化学元素。这个XML文件的部分内容列在本章后面“选择节点”示例中，其完整内容可以在本书网站的本章下载代码Elements.xml中找到。

表19-8列出了XPath执行的最常见操作。如果未特别说明，XPath查询示例就根据它操作的节点来选择。在必须有一个节点名称的地方，可以假定当前节点是XML文档中的<element>节点。

表19-8 XPath的常用操作

目的	XPath查询示例
选择当前节点	.

选择当前节点的父节点	..
选择当前节点的所有子节点	*
选择具有特定名称的所有子节点，这里是title	Title
选择当前节点的一个特性	@Type
选择当前节点的所有特性	@*
按照索引选择一个子节点，这里是第二个元素节点	element[2]
选择当前节点的所有文本节点	text()
选择当前节点的一个或多个孙子节点	element/text()
在文档中选择具有特定名称的所有节点，在这里是所有的mass节点	//mass
在文档中选择具有特定名称和特定父节点名称的所有节点，在这里父节点名称是element，节点名称是name	//element/name
选择值满足条件的节点，在这里，选择元素名为Hydrogen的元素	//element[name='Hydrogen']
选择特性值满足条件的节点，在此，Type特性的值是Noble Gas	//element[@Type='Noble Gas']

下面的“试一试”示例要创建一个小型应用程序，以执行许多预定义的查询，查看结果，并可以输入自己的查询。

试一试：选择节点： **Chapter19\XpathQuery\Elements.xml**

如前所示，这个示例使用新的XML文件Element.xml。可以从本书的网站上下载这个文件，或者输入下面的代码：

```
<?xml version="1.0"?>
<elements>
  <!--First Non-Metal-->
  <element Type="Non-Metal">
    <name>Hydrogen</name>
    <symbol>H</symbol>
    <number>1</number>
    <specification>
      <mass>1.007825</mass>
      <density>0.0899 g/cm3</density>
    </specification>
  </element>
  <!--First Noble Gas-->
  <element Type="Noble Gas">
    <name>Helium</name>
    <symbol>He</symbol>
    <number>2</number>
    <specification>
      <mass>4.002602</mass>
      <density>0.1785 g/cm3</density>
```

```

        </specification>
    </element>
    <!--First Halogen-->
    <element Type="Halogen">
        <name>Fluorine</name>
        <symbol>F</symbol>
        <number>9</number>
        <specification>
            <mass>18.998404</mass>
            <density>1.696 g/cm3</density>
        </specification>
    </element>
    <element Type="Noble Gas">
        <name>Neon</name>
        <symbol>Ne</symbol>
        <number>10</number>
        <specification>
            <mass>20.1797</mass>
            <density>0.901 g/cm3</density>
        </specification>
    </element>
</elements>

```

把这个XML文件保存为Elements.xml。一定要修改下述代码中文件的路径。这个示例是一个小型查询工具，可用于测试通过代码提供的XML上的不同查询。

按照下面的步骤创建一个具有查询功能的WPF应用程序。

(1) 创建一个新的WPF应用程序，命名为Xpath Query。

(2) 创建如图19-8所示的对话框。按照图中所示给控件命名，但按钮除外，它应命名为buttonExecute，将textBlock放到一个ScrollViewer控件中，并将其VerticalScrollBarVisibility属性设为Auto。



图19-8

(3) 进入代码视图，添加using指令。

(4) 接着，添加一个私有字段来保存文档，在构造函数中初始化它：

```
private XmlDocument document;  
public MainWindow()  
{
```

```

InitializeComponent();
document = new XmlDocument();
document.Load(@"C:\BegVCSharp\Chapter19\XML and Schema\Elem
}

```

(5) 在此需要使用几个帮助方法，以便在textBlockResult文本框中显示查询结果：

```

private void Update(XmlNodeList nodes)
{
    if (nodes == null || nodes.Count == 0)
    {
        textBlockResult.Text = "The query yielded no results";
        return;
    }
    string text = "";
    foreach (XmlNode node in nodes)
    {
        text = FormatText(node, text, "") + "\r\n";
    }
    textBlockResult.Text = text;
}

```

(6) 更新构造函数，以便在应用程序启动时显示XML文件的全部内容：

```

public MainWindow()
{

```

```

InitializeComponent();
document = new XmlDocument();
document.Load(@"C:\BegVCSharp\Chapter19\XML and Schema\El
Update(document.DocumentElement.SelectNodes("."));
}

```

(7) 将上一个“试一试”示例中的FormatText和AddAttributes方法复制粘贴到新项目中。

(8) 最后插入代码，执行用户在文本框中输入的内容：

```

private void buttonExecute_Click(object sender, RoutedEventArgs
{
    try
    {
        XmlNodeList nodes = document.DocumentElement.SelectNodes
        Update(nodes);
    }
    catch (Exception err)
    {
        textBlockResult.Text = err.Message;
    }
}

```

(9) 运行应用程序，在textBoxQuery文本框中输入下面的查询，以选择一个元素节点，其中包含文本为Hydrogen的节点。

```

element[name='Hydrogen']

```

示例的说明

`buttonExecute_Click`是执行查询的方法。因为我们不可能事先知道输入到`textBoxQuery`中的查询会生成一个还是多个节点，所以必须使用`SelectNodes`方法。该方法返回一个`XmlNodeList`对象，但如果所使用的查询是非法的，该方法就抛出一个与XPath相关的异常。

`Update`方法负责遍历`SelectNodes`选择出来的`XmlNodeList`的内容。它对每个节点调用前面示例中的`FormatText`，`FormatText`负责递归遍历节点树，创建可在`textBlockResult`控件中读取的文本。

在本章最后的练习中，读者需要执行其他许多XPath查询。在把它们输入XPathQuery应用程序中查看结果之前，尝试自己确定查询结果。

19.7 练习

(1) 修改“创建节点”示例中的插入方式，把值为1000+的特性Pages插入book节点。

(2) 确定下述XPath查询的结果，再把查询输入“选择节点”示例中的XPathQuery应用程序，来验证自己的结果。注意所有查询都在元素节点DocumentElement上执行。

```
//elements
element
element[@Type='Noble Gas']
//mass
//mass/..
element/specification[mass='20.1797']
element/name[text()='Neon']
Solution:
```

(3) 在许多Windows系统中，XML的默认查看器都是Web浏览器。如果使用Internet Explorer，在其中加载Elements.xml文件，就会看到美观的XML的格式化视图。在浏览器控件（而不是文本框）中显示查询的XML的效果为什么不理想？

(4) 使用Newtonsoft库将JSON转换成XML按钮（与本章所示的例子反向）。

附录A给出了练习答案。

19.8 本章要点

第20章 LINQ

本章内容：

- LINQ to XML
- LINQ提供程序
- LINQ查询语法
- LINQ方法语法
- Lambda表达式
- 对查询结果排序
- 聚合（Count, Sum, Min, Max, Average）
- SelectDistinctQuery
- 组合查询
- 连接

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 20 Code后，可以找到与本章示例对应的单独文件。

本章介绍Language Integrated Query（LINQ），这是C#语言的一个

扩展，可以将数据查询直接集成到编程语言本身中。

过去，完成这类任务需要编写大量的循环代码，而额外的处理甚至需要更多的代码，例如，排序或组合所找到的对象，因为数据源的不同会导致差异。而LINQ提供了一种可移式的、一致的方式，来查询、排序和分组许多不同种类的数据（XML、JSON、SQL数据库、对象集合、Web服务、企业目录等）。

首先以前一章的内容为基础，学习`system.xml.linq`名称空间添加的、用于创建XML的附加功能，然后进入LINQ的核心，学习如何使用查询语法、方法语法、Lambda表达式、排序、分组、连接相关的结果。

LINQ非常大，完整地介绍它的所有特性和方法已超出了本书的讨论范围。但本章将列举LINQ用户需要的所有运算符和语句类型的示例，并酌情给出深入探讨这些主题的资料。

20.1 使用LINQ to XML

LINQ to XML是用于XML的一组附加类，以使用LINQ to XML数据，如果以前没有使用LINQ，LINQ to XML还可以更方便地对XML执行某些操作。这里介绍LINQ to XML优于上一章讨论的XML DOM的几个特殊情形。

20.1.1 LINQ to XML函数构造方式

可在代码中用XML DOM创建XML文档，而LINQ to XML提供了一种更便捷的方式，称为函数构建方式（functional construction）。在这种方式中，构造函数的调用可以用反映XML文档结构的方式嵌套。下面的示例就使用函数构造方式建立了一个包含顾客和订单的简单XML文档。

试一试：**LINQ to XML:**
BegVCSharp_20_1_LinqtoXmlConstructors

按照下面的步骤在Visual Studio 2015中创建示例：

（1）在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_1_LinqToXmlConstructors。

(2) 打开主源文件Program.cs。

(3) 在Program.cs的开头处添加对System.Xml.Linq名称空间的引用，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
using static System.Console;
```

(4) 在Program.cs的Main()方法中添加如下代码：

```
static void Main(string[] args)
{
    XDocument xdoc = new XDocument(

    new XElement("customers",

    new XElement("customer",
```

```
new XAttribute("ID", "A"),
```

```
new XAttribute("City", "New York"),
```

```
new XAttribute("Region", "North America"),
```

```
new XElement("order",
```

```
new XAttribute("Item", "Widget"),
```

```
new XAttribute("Price", 100)
```

```
),
```



```
new XElement("order",
```

```
new XAttribute("Item", "Tire"),
```

```
new XAttribute("Price", 200)
```

```
)
```

```
),
```

```
new XElement("customer",
```

```
new XAttribute("ID", "B"),
```

```
new XAttribute("City", "Mumbai"),
```

```
new XAttribute("Region", "Asia"),
```

```
new XElement("order",
```

```
new XAttribute("Item", "Oven"),
```

```
new XAttribute("Price", 501)
```

```
)
```

```
)
```

```
)
```

```
);
```

```
WriteLine(xdoc);
```

```
Write("Program finished, press Enter/Return to continue:");
```

```
ReadLine();
```

```
}
```

(5) 编译并执行程序（按下F5键即可开始调试），输出结果如下所示：

```
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>

Program finished, press Enter/Return to continue:
```

输出屏幕上显示的XML文档包含前面示例中顾客/订单数据的一个简化版本。注意，XML文档的根元素是<customers>，它包含两个嵌套的<customer>元素，这两个元素又包含许多嵌套的<order>元素。<customer>元素具有两个特性：<City>和<Region>，<order>元素也有两个特性：<Item>和<Price>。

按下回车键，以便结束程序，关闭控制台屏幕。如果使用Ctrl+F5组合键（启动时不使用调试功能），就需要按回车键两次。

示例的说明

第一步是引用System.Xml.Linq名称空间。本章的所有XML示例都要求把这行代码添加到程序中：

```
using System.Xml.Linq;
```

在创建项目时，System.Linq名称空间是默认包含的，但不包含

System.Xml.Linq名称空间，所以必须显式地添加这行代码。

接着调用LINQ to XML构造函数XDocument()、XElement()和XAttribute()，它们彼此嵌套，如下所示：

```
XDocument xdoc = new XDocument(  
    new XElement("customers",  
        new XElement("customer",  
            new XAttribute("ID", "A"),  
            ...  
        )  
    )  
);
```

注意，这些代码看起来类似于XML本身，即文档包含元素，每个元素又包含特性和其他元素。下面依次分析这些构造函数：

- XDocument()：在LINQ to XML构造函数层次结构中，最高层的对象是XDocument()，它表示完整的XML文档，在代码中如下所示：

```
static void Main(string[] args)  
{  
    XDocument xdoc = new XDocument(  
        ...  
    );  
}
```

在前面的代码段中，省略了XDocument()的参数列表，因此可以看到XDocument()调用在何处开始和结束。与所有LINQ to XML构造函数一样，XDocument()也把对象数组（object[]）作为它的参数之一，以便给它传送其他构造函数创建的其他多个对象。在这个程序中调用的所有其他构造函数都是XDocument()构造函数的参数。这个程序传送的第一个也是唯一一个参数是XElement()构造函数。

- XElement(): XML文档必须有一个根元素，所以大多数情况下，XDocument()的参数列表都以一个XElement对象开头。XElement()构造函数把元素名作为字符串，其后是包含在该元素中的一个XML对象列表。本例中的根元素是customers，它又包含一个customer元素列表：

```
        new XElement("customers",  
            new XElement("customer",  
                ...  
            ),  
            ...  
        )
```

customer元素不包含其他XML元素，只包含3个XML特性，它们用XAttribute()构造函数构建。

- XAttribute(): 这里给customer元素添加了3个XML特性：ID、City和Region：

```
        new XAttribute("ID", "A"),  
        new XAttribute("City", "New York"),  
        new XAttribute("Region", "North America"),
```

根据定义，XML特性是一个XML叶节点，它不包含其他XML节点，所以XAttribute()构造函数的参数只有特性的名称和值。本例中生成的3个特性是ID="A"、City="New York"和Region="North America"。

- 其他LINQ to XML构造函数：这个程序中没有调用它们，但所有XML节点类型都有其他LINQ to XML构造函数，例如，

XDeclaration()用于XML文档开头的XML声明，XComment()用于XML注释等。这些构造函数不太常用，但如果需要它们来精确控制XML文档的格式化，就可以使用它们。

下面继续解释第一个示例：在customer元素的ID、City和Region特性后面再添加两个子order元素：

```
new XElement("order=",
    new XAttribute("Item", "Widget"),
    new XAttribute("Price", 100)
),
new XElement("order",
    new XAttribute("Item", "Tire"),
    new XAttribute("Price", 200)
)
```

这些order元素都有两个特性：Item和Price，但没有子元素。

接着将XDocument的内容显示在控制台屏幕上：

```
WriteLine(xdoc);
```

这行代码使用XDocument()的默认方法ToString()输出XML文档的文本。

最后暂停屏幕，以便查看控制台输出，再等待用户按下回车键：

```
Write("Program finished, press Enter/Return to continue:");
ReadLine();
```

程序退出Main()方法后，就结束程序。

20.1.2 处理XML片段

与一些XML DOM不同，LINQ to XML处理XML片段（部分或不完整的XML文档）的方式与处理完整的XML文档几乎完全相同。在处理片段时，只需将XElement（而不是XDocument）当作顶级XML对象。

注意： 唯一的限制是不能添加某些比较深奥的、只能应用于XML文档或XML片段的XML节点类型，例如，XComment应用于XML注释，XDeclaration应用于XML文档声明，XProcessingInstruction用于XML处理指令。

下面的示例会加载、保存和处理XML元素及其子节点，这与处理XML文档一样。

试一试：处理XML片段： **BegVCSharp_20_2_XMLFragments**

按照下面的步骤在Visual Studio 2015中创建示例：

（1）在C:\BegVCSharp\Chapter20目录中修改上一个示例或者创建一个新的控制台应用程序BegVCSharp_20_2_XMLFragments。

(2) 打开主源文件Program.cs。

(3) 在Program.cs开头处添加对System.Xml.Linq名称空间的引用，如下所示：

```
using System;
using System.Collections.Generic;
using System.Xml.Linq;
using System.Text;
using static System.Console;
```

如果正在修改上一个示例，则已经引用了这个名称空间。

(4) 把上一个示例中的XML元素（不包含XML文档构造函数）添加到Program.cs的Main()方法中：

```
static void Main(string[] args)
{
    XElement xcust =

        new XElement("customers",

            new XElement("customer",
```

```
new XAttribute("ID", "A"),
```

```
new XAttribute("City", "New York"),
```

```
new XAttribute("Region", "North America"),
```

```
new XElement("order",
```

```
new XAttribute("Item", "Widget"),
```

```
new XAttribute("Price", 100)
```

```
),
```

```
new XElement("order",
```

```
new XAttribute("Item", "Tire"),
```

```
new XAttribute("Price", 200)
```

```
)
```

```
),
```

```
new XElement("customer",
```

```
new XAttribute("ID", "B"),
```

```
new XAttribute("City", "Mumbai"),
```

```
new XAttribute("Region", "Asia"),
```

```
new XElement("order",
```

```
new XAttribute("Item", "Oven"),
```

```
new XAttribute("Price", 501)
```

```
)
```

)

)

;

(5) 在上一步添加了XML元素构造函数代码后，添加下面的代码，以便保存、加载和显示XML元素：

```
        string xmlFileName =  
        @"c:\BegVCSharp\Chapter20\BegVCSharp_20_2_XMLFragments\frag  
        xcust.Save(xmlFileName);  
        XElement xcust2 = XElement.Load(xmlFileName);  
        WriteLine("Contents of xcust:");  
        WriteLine(xcust);  
        Write("Program finished, press Enter/Return to continu  
        ReadLine();  
    }
```

注意： xmlFileName是一个绝对路径；读者的文件夹结构可能不同，此时，应该调整路径，以反映自己计算机上实际的文件夹路径。

(6) 编译并执行程序（按下F5键即可开始调试），控制台窗口中的输出结果如下所示：

Contents of XElement xcust2:

```
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>
```

Program finished, press Enter/Return to continue:

按下回车键，以便结束程序，关闭控制台屏幕。如果使用Ctrl+F5组合键（启动时不使用调试功能），就需要按下回车键两次。

示例的说明

XElement和XDocument都继承自LINQ to XML类XContainer，它实现了一个可以包含其他XML节点的XML节点。这两个类都实现了Load()和Save()方法，因此，可以在LINQ to XML的XDocument上执行的大多数操作都可以在XElement实例及其子元素上执行。

这里只创建了一个XElement实例，它的结构与前面示例中的

XDocument相同，但不包含XDocument。这个程序的所有操作处理的都是XElement片段。

XElement还支持Load()和Parse()方法，可以分别从文件和字符串中加载XML。

20.2 LINQ提供程序

LINQ to XML只是LINQ提供程序的一个例子。Visual Studio 2015和.NET Framework 4.5有许多内置LINQ提供程序，为不同类型的数据提供了查询解决方案：

- **LINQ to Objects**：对任何类型的C#内存中对象提供查询，比如数组、列表和其他集合类型。上一章的所有例子都使用了LINQ to Objects。也可以把在本章学到的技术应用于LINQ的所有变体。
- **LINQ to XML**：如前所述，它使用与其他LINQ变体相同的语法和通用查询机制，来创建和操纵XML文档。
- **LINQ to Entities**：Entity Framework是.NET 4中最新的数据接口类，Microsoft建议使用它进行新的开发工作。本章给Visual C#项目添加一个ADO.NET Entity Framework数据源，然后使用LINQ to Entities查询它。
- **LINQ to Data Set**：DataSet对象在.NET Framework的第1版引入。这个LINQ变体支持使用LINQ方便地查询旧.NET数据。
- **LINQ to SQL**：这是另一个LINQ接口，取代了LINQ to Entities。
- **PLINQ**：PLINQ是并行LINQ，用并行编程库扩展了LINQ to Objects，可以拆分查询，让它们在多核处理器上同时执行。
- **LINQ to JSON**：包括在上一章使用的Newtonsoft包中，这个库支持使用与其他LINQ变体相同的语法和通用查询机制，来创建和操纵JSON文档。

有这么多种类的LINQ，一本入门书籍不可能覆盖所有的内容，但

是其语法和方法适用于所有的种类。接下来使用LINQ to Objects提供程序介绍LINQ查询语法。

20.3 LINQ查询语法

下面的示例使用LINQ创建了一个查询，以便在一个简单的内存对象数组中查找一些数据，并输出到控制台上。

试一试：第一个LINQ程序：

BegVCSharp_20_3_QuerySyntax\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_3_QuerySyntax，然后打开主源文件Program.cs。

（2）注意，Visual Studio 2015默认在Program.cs中包含System.Linq名称空间：

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
using System.Text;  
using static System.Console;
```

```
using System.Threading.Text;
```

(3) 在Program.cs的Main()方法中添加如下代码:

```
static void Main(string[] args)
```

```
{
```

```
    string[] names = { "Alonso", "Zheng", "Smith", "Jones",
```

```
"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Sam
```

```
    var queryResults =
```

```
        from n in names
```

```
        where n.StartsWith("S")
```

```
        select n;
```

```
WriteLine("Names beginning with S:");
```

```
foreach (var item in queryResults) {
```

```
    WriteLine(item);
```

```
}
```

```
Write("Program finished, press Enter/Return to continue
```

```
ReadLine());
```

```
}
```

（4）编译并运行程序（按下F5键即可开始调试），列表中的名称以S开头，按照它们在数组中的声明顺序排列，如下所示。

```
Names beginning with S:
Smith
Smythe
Small
Singh
Samba
Program finished, press Enter/Return to continue:
```

按下回车键，结束程序，关闭控制台屏幕。如果使用Ctrl+F5组合键（启动时不使用调试功能），就需要按下回车键两次，这会结束程序的运行。

示例的说明

第一步是引用System.Linq名称空间，这在创建项目时由Visual Studio 2015自动完成：

```
using System.Linq;
```

所有的基本底层系统都支持System.Linq名称空间中用于LINQ的类。如果在Visual Studio 2015外部创建C#源文件或编辑以前版本创建的项目，就必须手动添加using System.Linq指令。

下一步创建一些数据，在本例中就是声明并初始化names数组：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smyth",  
"Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fat
```

这些数据很少，很适合用于查询结果比较明显的示例。程序的下一部分是真正的LINQ查询语句：

```
var queryResults =  
    from n in names  
    where n.StartsWith("S")  
    select n;
```

这是一个看起来比较古怪的语句。它不像是C#语言，实际上from...where...select语法类似于SQL数据库查询语言。但这个语句不是SQL，而是C#，在Visual Studio 2015中输入这些代码时，from、where和select会突出显示为关键字，这个古怪的语法对编译器而言是完全正确的。

这个程序中的LINQ查询语句使用了LINQ声明性查询语法：

```
var queryResults =  
    from n in names  
    where n.StartsWith("S")  
    select n;
```

该语句包括4个部分：以var开头的结果变量声明，使用查询表达式给该结果变量赋值，查询表达式包含from子句、where子句和select子句。下面逐一介绍它们。

20.3.1 用var关键字声明结果变量

LINQ查询首先声明一个变量，以包含查询的结果，这通常是用var关键字声明一个变量来完成的：

```
var queryResult =
```

var是C#中的一个新关键字，用于声明一般的变量类型，特别适于包含LINQ查询的结果。var关键字告诉C#编译器，根据查询推断结果的类型。这样，就不必提前声明从LINQ查询返回的对象类型了——编译器会推断出该类型。如果查询返回多个条目，该变量就是查询数据源中的一个对象集合（从技术角度看，它并不是一个集合，只是看起来像是集合而已）。

注意： 如果希望了解细节，查询结果将是实现了IEnumerable<T>接口的类型。IEnumerable后面带字母T的尖括号（<T>）表示这是一个泛型类型。泛型详见第12章。

在上例中，编译器创建了一个特殊的LINQ数据类型，它提供了字符串的有序列表（因为数据源是一个字符串集合）。

另外，queryResult名称是随意指定的，可以把结果命名为任何名称，例如，namesBeginningWithS或者在程序中有意义的其他名称。

20.3.2 指定数据源：from子句

LINQ查询的下一部分是from子句，它指定了要查询的数据：

```
from n in names
```

本例中的数据源是前面声明的字符串数组names。变量n只是数据源中某一元素的代表，类似于foreach语句后面的变量名。指定from子句，就可以只查找集合的一个子集，而不必迭代所有的元素。

说到迭代，LINQ数据源必须是可枚举的——即必须是数组或集合，以便从中选择一个或多个元素。

注意：可枚举意味着数据源必须支持IEnumerable<T>接口，所有C#数组或项集合都支持这个接口。

数据源不能是单个值或对象，例如，单个int变量。如果只有一项，就没必要查询了。

20.3.3 指定条件：where子句

在LINQ查询的下一部分，可以用where子句指定查询的条件，如下所示：

```
where n.StartsWith("S")
```

可以在where子句中指定能应用于数据源中各元素的任意布尔（true或false）表达式。实际上，where子句是可选的，甚至可以忽略，但大

多数情况下，都要指定where条件，把结果限制为我们需要的数据。where子句称为LINQ中的限制运算符，因为它限制了查询的结果。

这个示例指定name字符串以字母S开头，还可以给字符串指定其他条件，例如，长度超过10（where n.Length>10）或者包含Q（where n.Contains（"Q"））。

20.3.4 选择元素：select子句

最后，select子句指定结果集中包含哪些元素。select子句如下所示：

```
select n;
```

select子句是必需的，因为必须指定结果集中有哪些元素。这个结果集并不是很有趣，因为在结果集的每个元素中都只有一项name。如果结果集中有比较复杂的对象，使用select子句的有效性就比较明显，不过我们还是首先完成这个示例。

20.3.5 完成：使用foreach循环

现在输出查询的结果。与把数组用作数据源一样，像这样的LINQ查询结果是可以枚举的，即可以用foreach语句迭代结果：

```
WriteLine("Names beginning with S:");  
foreach (var item in queryResults) {  
    WriteLine(item);  
}
```

```
}
```

在本例中，匹配了5个名称：Smith、Smythe、Small、Singh和Samba，所以它们会显示在foreach循环中。

20.3.6 延迟执行的查询

foreach循环实际上并不是LINQ的一部分，它只是迭代结果。虽然foreach结构并不是LINQ的一部分，但它是实际执行LINQ查询的代码。查询结果变量仅保存了执行查询的一个计划，在访问查询结果之前，并没有提取LINQ数据，这称为查询的延迟执行或迟缓执行。生成结果序列（即列表）的查询都要延迟执行。

现在回过头来看看代码。由于输出了结果，所以程序结束：

```
Write("Program finished, press Enter/Return to continue:");  
ReadLine();
```

这些代码仅确保在按下一个键（甚至可以按下F5键，而不是Ctrl+F5组合键）之前，控制台程序的结果始终显示在屏幕上。在大多数其他LINQ示例中也使用这种结构。

20.4 LINQ方法语法

使用LINQ完成同一任务有多种方式，这与编程时一样。如前所述，前面的示例是用LINQ查询语法编写的，下一个示例是用LINQ的方法语法（也称为显式语法，但这里使用“方法语法”这个术语）编写的相同程序。

20.4.1 LINQ扩展方法

LINQ实现为一系列扩展方法，用于集合、数组、查询结果和其他实现了IEnumerable<T>接口的对象。在Visual Studio IntelliSense特性中可以看到这些方法。例如，在Visual Studio 2015中打开FirstLINQquery程序中的Program.cs文件，在name数组的下面输入对该数组的一个新引用：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smyth",  
"Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fat  
names.
```

输入names后面的句点后，就会看到Visual Studio IntelliSense列出的可用于names的方法。

Where<T>方法与大多数其他方法都是扩展方法（在Where<T>方法的右边显示了一个文档说明，它以extension开头）。因为如果在顶部注释掉了using System.Linq指令，Where<T>、Union<T>、Take<T>和大多数其他方法就会从列表中消失。上一个示例使用的from..where..select查询表达式由C#编译器转换为这些方法的一系列调用。使用LINQ方法语法时，就直接调用这些方法。

20.4.2 查询语法和方法语法

查询语法是在LINQ中编写查询的首选方式，因为它一般更容易理解，最常见的查询使用它们也更简单。但是，一定要基本了解方法语法，因为一些LINQ功能不能通过查询语法来使用，或者使用方法语法比较简单。

注意： Visual Studio 2015联机帮助建议尽量使用查询语法，仅在需要时使用方法语法。

本章主要使用查询语法，但会指出需要方法语法的场合，并说明如何使用方法语法来解决问题。

大多数使用方法语法的LINQ方法都要求传送一个方法或函数，来计算查询表达式。方法/函数参数以委托形式传送，它一般引用一个匿名方法。

LINQ很容易完成这个传送任务。使用Lambda表达式就可以创建方

法/函数，它以优雅的方式封装委托。

20.4.3 Lambda表达式

Lambda表达式很容易随时创建在LINQ查询中使用的方法。它使用`=>`操作符，它在一行代码中声明方法的参数后跟方法的逻辑。

注意：“Lambda表达式”这个词来自微积分，这是编程语言理论中的一个重要的数学领域。如果读者擅长数学，可以查一下。幸好，在C#中使用Lambda不需要数学！

例如下面的Lambda表达式：

```
n => n<0
```

这个语句声明了一个带单一参数`n`的方法。如果`n`小于0，该方法就返回`true`，否则返回`false`。这是非常简单的。不需要方法名、返回语句，也不需要花括号将任何代码括起来。

像这样返回`true/false`值是LINQ的Lambda表达式中的方法常用的方式，但这不是必需的。例如，下面的Lambda表达式创建了一个方法，它返回两个变量的和。这个Lambda表达式使用了多个参数：

```
(a, b) => a + b
```

这个语句声明一个带两个参数`a`和`b`的方法。方法逻辑返回`a`和`b`的

和。不必声明a和b的类型是什么。它们可以是int、double或string。C#编译器会推断出类型。

最后考虑下面的Lambda表达式：

```
n => n.StartsWith("S")
```

如果n以字母S开头，这个方法就返回true，否则返回false。下一个示例将更清楚地说明这一点。

试一试：使用**LINQ**方法语法和**Lambda**表达式：
BegVCSharp_20_4_MethodSyntax\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）可以修改FirstLINQquery示例，或在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_4_MethodSyntax。打开主源文件Program.cs。

（2）Visual Studio 2015会自动在Program.cs中包含Linq名称空间：

```
using System.Linq;
```

（3）在Program.cs的Main()方法中添加如下代码：

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones"
```

```

"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Sam
    var queryResults = names.Where(n => n.StartsWith("S"))
    WriteLine("Names beginning with S:");
    foreach (var item in queryResults) {
        WriteLine(item);
    }
    Write("Program finished, press Enter/Return to continu
    ReadLine();
}

```

（4）编译并执行程序（可按下F5键）。结果也是以S开头的names列表，且按照它们在数组中声明的顺序排列，如下所示：

```

Names beginning with S:
Smith
Smythe
Small
Singh
Samba
Program finished, press Enter/Return to continue:

```

示例的说明

与前面一样，Visual Studio 2015会自动引用System.Linq名称空间：

```
using System.Linq;
```

再次声明和初始化names数组，创建相同的源数据：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smyth",  
    "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah"
```

LINQ查询是不同的，它现在是Where()方法的调用，而不是查询表达式：

```
var queryResults = names.Where(n => n.StartsWith("S"));
```

C#编译器把Lambda表达式`n=>n.StartsWith("S")`编译为一个匿名方法，Where()在names数组的每个元素上执行这个方法。如果Lambda表达式给某个元素返回true，该元素就包含在Where()返回的结果集中。C#编译器从输入数据源（这里是names数组）的定义中推断，该Where()方法应把string作为每个元素的输入类型。

许多工作都是在一行代码中完成的。对于像这样最简单的查询，方法语法要比查询语法更加简短，因为不需要from或select子句，但大多数查询都比这更复杂。

示例的剩余部分与前面的代码相同——在foreach循环中显示查询的结果，并暂停输出，以便在程序执行完毕前看到结果：

```
foreach (var item in queryResults) {  
    WriteLine(item);  
}  
Write("Program finished, press Enter/Return to continue:");  
ReadLine();
```

这里不重复说明这些代码行，因为本章第一个示例后面的“示例的说明”已经解释过了。下面继续研究如何使用LINQ的更多功能。

20.5 排序查询结果

用where子句（或者Where()方法调用）找到了感兴趣的数据后，LINQ还可以方便地对得到的数据执行进一步处理，例如，重新排列结果的顺序。下面的示例将按字母顺序给第一个查询的结果排序。

试一试：给查询结果排序：

BegVCSharp_20_5_OrderQueryResults\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）可以修改QuerySyntax示例，或者在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序项目BegVCSharp_20_5_OrderQueryResults。

（2）打开主源文件Program.cs。与以前一样，Visual Studio 2015会自动在Program.cs中包含using System. Linq;名称空间指令。

（3）在Program.cs的Main()方法中添加如下代码：

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones"
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Sam"
```

```

        var queryResults =
            from n in names
            where n.StartsWith("S")
            orderby n
            select n;
        WriteLine("Names beginning with S ordered alphabetically");
        foreach (var item in queryResults) {
            WriteLine(item);
        }
        Write("Program finished, press Enter/Return to continue:");
        ReadLine();
    }
}

```

（4）编译并执行程序。结果是以S开头的names列表，且按字母顺序排序，如下所示：

```

Names beginning with S:
Samba
Singh
Small
Smith
Smythe
Program finished, press Enter/Return to continue:

```

示例的说明

这个程序与前一个程序几乎相同，只是在查询语句中增加了一行代码：

```
var queryResults =  
    from n in names  
    where n.StartsWith("S")  
    orderby n  
  
    select n;
```

20.6 orderby子句

orderby子句如下所示：

```
orderby n
```

与where子句一样，orderby子句也是可选的。只要添加一行，就可以对任意查询的结果排序，而不使用LINQ时，根据选择实现的排序算法，需要额外编写至少几行代码，还可能添加几个方法或集合来存储重新排序的结果。如果有多个需要排序的类型，就需要为每个类型实现一系列排序方法。而使用LINQ不需要做这些工作，只需要在查询语句中添加一条子句即可。

orderby子句默认为升序（A到Z），但可以添加descending关键字，以便指定降序（Z到A）：

```
orderby n descending
```

这会使示例的结果变成：

```
Smythe  
Smith  
Small  
Singh  
Samba
```

另外，可以按照任意表达式进行排序，而不必重新编写查询。例

如，要按照姓名中的最后一个字母排序，而不是按一般的字母顺序排序，就只需要添加如下`orderby`子句：

```
orderby n.Substring(n.Length - 1)
```

结果如下：

Samba

Smythe

Smith

Singh

Small

注意： 最后一个字母按字母顺序排序（a，e，h，h，l）。但这个执行过程依赖于实现方式，即无法保证除了`orderby`子句中指定的内容之外的其他字母的顺序。由于仅考虑最后一个字母，因此在本例中，**Smith**在**Singh**的前面。

20.7 查询大型数据集

LINQ语法非常好，但其作用是什么？我们只要查看源数组，就可以看出需要的结果，为什么要查询这种一眼就能看出结果的数据源呢？如前所述，有时查询的结果不那么明显。在下面的示例中，就创建了一个非常大的数字数组，并用LINQ查询它。

试一试：查询大型数据集：

BegVCSharp_20_6_LargeNumberQuery\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_6_Large NumberQuery。与以前一样，创建项目时，Visual Studio 2015会自动在Program.cs中包含Linq名称空间。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using static System.Console;
```

（2）在Main()方法中添加如下代码：

```
static void Main(string[] args)
{
    int[] numbers = GenerateLotsOfNumbers(12045678);

    var queryResults =

        from n in numbers

        where n<1000

        select n

    ;

    WriteLine("Numbers less than 1000:");
```



```
foreach (var item in queryResults)
```

```
{
```

```
    WriteLine(item);
```

```
}
```

```
Write("Program finished, press Enter/Return to continue
```

```
ReadLine());
```

```
}
```

(3) 添加如下方法，生成一个随机数列表：

```
private static int[] GenerateLotsOfNumbers(int count)
{
    Random generator = new Random(0);
    int[] result = new int[count];
    for (int i = 0; i < count; i++)
    {
        result[i] = generator.Next();
    }
    return result;
}
```

(4) 编译并执行程序。结果是一个小于1000的数字列表，如下所示：

Numbers less than 1000:

714

24

677

350

257

719

584

Program finished, press Enter/Return to continue:

示例的说明

与前面一样，第一步是引用System.Linq名称空间，这是在创建项目时由Visual Studio 2015自动引用的：

```
using System.Linq;
```

接着创建一些数据，本例中是创建并调用GenerateLotsOfNumbers()方法：

```
int[] numbers = GenerateLotsOfNumbers(12345678);  
private static int[] GenerateLotsOfNumbers(int count)  
{  
    Random generator = new Random(0);  
    int[] result = new int[count];  
    for (int i = 0; i<count; i++)  
    {  
        result[i] = generator.Next();  
    }  
    return result;  
}
```

这不是一个小数据集，数组中有1200万个数字！在本章最后的一个练习中，需要修改传送给GenerateLotsOfNumbers()方法的size参数，生成数量不同的随机数，看看这会对查询结果有什么影响。在做练习时会看到，这里的size参数12 345 678非常大，足以生成一些小于1 000的随机数，从而获得为第一个查询显示的结果。

数值应随机分布在有符号的整数范围内（从0到超过20亿）。用种

子值0创建随机数生成器，可以确保每次创建相同的随机数集合，这是可以重复的，所以会获得与此处相同的查询结果，但在尝试一些查询之前，并不知道查询结果是什么。而LINQ使这些查询很容易编写。

查询语句本身类似于前面用于names数组的查询，也是选择某些满足条件的数字（这里是数字小于1 000）：

```
var queryResults =  
    from n in numbers  
    where n<1000  
    select n
```

这次不需要orderby子句，但处理时间略长（对于这个查询，处理时间的变化不太明显，但下一个示例会改变选择条件，处理时间的变化就比较明显了）。

用foreach语句输出查询的结果，与前面的示例相同：

```
WriteLine("Numbers less than 1000:");  
foreach (var item in queryResults) {  
    WriteLine(item);  
}
```

同样，将结果输出到控制台上，并读取一个字符以暂停输出：

```
Write("Program finished, press Enter/Return to continue:");  
ReadLine();
```

后面所有的示例都有暂停代码，但不再列出，因为每个示例的暂停代码都相同。

使用LINQ，可以很容易地修改查询条件，以便演示数据集的不同特性。但是，根据查询返回的结果数，每次都输出所有的结果是没有意义的。下一节将说明LINQ提供的聚合运算符是如何处理这种情况的。

20.8 使用聚合运算符

查询给出的结果常常超出了我们的期望。如果要修改大数查询程序的条件，只需要列出大于1 000的数字，而不是小于1 000的数字，这会得到非常多的查询结果，数字会不停地显示出来。

LINQ提供了一组聚合运算符，可用于分析查询结果，而不必迭代所有结果。表20-1列出的聚合运算符是数字结果集最常用的运算符，例如，大数查询的结果就常用这些聚合运算符，如果读者使用过数据库查询语言（如SQL），就会十分熟悉这些运算符。

表20-1 数字结果的聚合运算符

运算符	说明
Count()	结果的个数
Min()	结果中的最小值
Max()	结果中的最大值
Average()	数字结果的平均值
Sum()	所有数字结果的总和

还有更多的聚合运算符，如Aggregate()，它们可以执行代码，并允许编写自己的聚合函数。但是，这些都用于高级用户，超出了本书的讨论范围。

注意：聚合运算符返回一个简单的标量类型，而不是一系列结果，所以使用它们会强制立即执行查询，而不是延迟执行。

下面的示例修改大数查询，并使用聚合运算符和LINQ分析大数查询的大于版本中的结果集。

试一试：数字聚合运算符：

BegVCSharp_20_7_NumericAggregates\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

(1) 这个示例可以修改前面的LargeNumberQuery示例，或在C:\BegVCSharp\Chapter20目录中创建一个新的控制台项目BegVCSharp_20_7_NumericAggregates。

(2) 与以前一样，创建项目时，Visual Studio 2015会自动在Program.cs中包含Linq名称空间。只需要修改Main()方法，如下面的代码和本示例其余部分所示。与上一个例子一样，这个查询也不使用orderby子句。但是where子句中的条件与前一个例子相反（数字应大于1 000（ $n > 1000$ ），而不是小于1 000）。

```
static void Main(string[] args)
{
    int[] numbers = GenerateLotsOfNumbers(12345678);
```

```
WriteLine("Numeric Aggregates");
```

```
var queryResults =  
    from n in numbers  
    where n > 1000  
  
    select n  
    ;  
WriteLine("Count of Numbers > 1000");
```

```
WriteLine(queryResults.Count());
```

```
WriteLine("Max of Numbers > 1000");
```

```
WriteLine(queryResults.Max());
```



```
WriteLine("Min of Numbers > 1000");
```

```
WriteLine(queryResults.Min());
```

```
WriteLine("Average of Numbers > 1000");
```

```
WriteLine(queryResults.Average());
```

```
WriteLine("Sum of Numbers > 1000");
```

```
WriteLine(queryResults.Sum(n => (long) n));
```

```
Write("Program finished, press Enter/Return to continu  
ReadLine());
```

```
}
```

(3) 添加上一个示例中使用的GenerateLotsOfNumbers()方法（如不存在）：

```
private static int[] GenerateLotsOfNumbers(int count)
{
    Random generator = new Random(0);
    int[] result = new int[count];
    for (int i = 0; i < count; i++)
    {
        result[i] = generator.Next();
    }
    return result;
}
```

(4) 编译并执行，显示个数、最小值、最大值和平均值，如下所示：

```
Numeric Aggregates
Count of Numbers > 1000
12345671
Maximum of Numbers > 1000
2147483591
Minimum of Numbers > 1000
1034
Average of Numbers > 1000
1073643807.50298
```

```
Sum of Numbers > 1000
```

```
13254853218619179
```

```
Program finished, press Enter/Return to continue:
```

这个查询生成的结果数量超过上一个例子（超过1200万）。在这个结果集上使用`orderby`，对性能会有很显著的影响。结果集中的最大值超过20亿，最小值刚刚大于1 000。平均值大约是10亿，接近数字范围的中间值。看来，`Random()`函数可以生成均匀分布的数字。

示例的说明

程序的第一部分与上一个例子完全相同，也是引用`System.Linq`名称空间，然后用`GenerateLotsOfNumbers()`方法生成源数据：

```
int[] numbers = GenerateLotsOfNumbers(12345678);
```

查询也与上一个例子相同，只是把`where`条件从小于改为大于：

```
var queryResults =  
    from n in numbers  
    where n > 1000
```

```
select n;
```

如前所述，使用大于条件的这个查询生成的结果远远多于小于查询（对这个数据集而言）。使用聚合运算符可以分析查询结果，而不必输

出每个结果，或者在foreach循环中比较它们。每个聚合运算符都类似于一个可在结果集上调用的方法，也类似于在集合类型上调用的方法。

下面看一下每个聚合运算符的用法：

- Count():

```
WriteLine("Count of Numbers > 1000");  
WriteLine(queryResults.Count());
```

Count（）返回查询结果中的行数，在这个例子中是12 345 671行

- Max():

```
WriteLine("Max of Numbers > 1000");  
WriteLine(queryResults.Max());
```

Max（）返回查询结果中的最大值，在这个例子中是大于20亿的一个数2 147 483 591，它非常接近int的最大值（int.MaxValue或2 147 483 647）。

- Min():

```
WriteLine("Min of Numbers > 1000");  
WriteLine(queryResults.Min());
```

Min()返回查询结果中的最小值，在这个例子中是1 034。

- Average():

```
WriteLine("Average of Numbers > 1000");
```

```
WriteLine(queryResults.Average());
```

Average()返回查询结果中的平均值，在这个例子中是1 073 643 807.502 98，它非常接近1 000到20亿的值范围的中间值。对于随机的大数而言，这个中间值没有什么意义，但说明了可以对查询结果进行分析。本章最后一部分将使用这些运算符对面向业务的数据进行更贴近实际的分析。

- Sum():

```
WriteLine("Sum of Numbers > 1000");  
WriteLine(queryResults.Sum(n => (long) n));
```

在此给Sum()方法调用传送了Lambda表达式n=> (long) n，以获得所有数字的总和。与Count()、Min()、Max()等相同，Sum()有一个无参数的重载版本，但使用Sum()方法的这个版本会导致溢出错误，因为数据集中的数字太多，它们的总和太大，不能放在Sum()方法的无参数重载版本返回的标准的32位int中。Lambda表达式允许把Sum()方法的结果转换为64位长整数，它可以保存超过 13×10^{15} 的数字13 254 853 218 619 179，而不出现溢出。Lambda表达式允许方便地执行这个转换。

注意： Count()返回32位int。LINQ还提供了一个LongCount()方法，它在64位整数中返回查询结果的个数。但有一个特殊情况：如果需要数字的64位版本，所有其他运算符都需要一个Lambda表达式或调用一个转换方法。

20.9 单值选择查询

在SQL数据查询语言中，我们熟悉的另一类查询是SELECT DISTINCT查询，该查询可搜索数据中的唯一值，也就是说，值是不重复的。这是使用查询时的一个常见需求。

假定需要在前面示例使用的顾客数据中查找不同的区域，由于在这些数据中没有单独的区域列表，所以需要从顾客列表中找出唯一的、不重复的区域列表。LINQ提供了Distinct()方法，以便找出这些数据，如下面的示例所示。

试一试：投影——单值选择查询：

BegVCSharp_20_8_SelectDistinctQuery\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_8_SelectDistinctQuery。

（2）输入如下代码，创建Customer类，初始化customers列表（List<Customer>customers）：

```
class Customer
{
```

```

    public string ID { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
    public string Region { get; set; }
    public decimal Sales { get; set; }
    public override string ToString()
    {
        return "ID: " + ID + " City: " + City +
            " Country: " + Country +
            " Region: " + Region +
            " Sales: " + Sales;
    }
}

class Program
{
    static void Main(string[] args)
    {
        List<Customer> customers = new List<Customer> {
            new Customer { ID="A", City="New York", Country="North America", Sales=9999},
            new Customer { ID="B", City="Mumbai", Country="India", Sales=8888},
            new Customer { ID="C", City="Karachi", Country="Pakistan", Sales=7777},
            new Customer { ID="D", City="Delhi", Country="India", Sales=6666},
            new Customer { ID="E", City="São Paulo", Country="Brazil", Sales=5555}
        };
    }
}

```

```
Region="South America", Sales=5555 },
    new Customer { ID="F", City="Moscow", Country="Ru
Region="Europe", Sales=4444 },
    new Customer { ID="G", City="Seoul", Country="Kor
Region="Asia", Sales=3333 },
    new Customer { ID="H", City="Istanbul", Country="
Region="Asia", Sales=2222 },
    new Customer { ID="I", City="Shanghai", Country="
Region="Asia", Sales=1111 },
    new Customer { ID="J", City="Lagos", Country="Nig
Region="Africa", Sales=1000 },
    new Customer { ID="K", City="Mexico City", Countr
Region="North America", Sales=2000 },
    new Customer { ID="L", City="Jakarta", Country="I
Region="Asia", Sales=3000 },
    new Customer { ID="M", City="Tokyo", Country="Jap
Region="Asia", Sales=4000 },
    new Customer { ID="N", City="Los Angeles", Countr
Region="North America", Sales=5000 },
    new Customer { ID="O", City="Cairo", Country="Egy
Region="Africa", Sales=6000 },
    new Customer { ID="P", City="Tehran", Country="Ir
Region="Asia", Sales=7000 },
    new Customer { ID="Q", City="London", Country="UK
Region="Europe", Sales=8000 },
    new Customer { ID="R", City="Beijing", Country="C
Region="Asia", Sales=9000 },
```



```
        new Customer { ID="S", City="Bogotá", Country="Colombia",  
Region="South America", Sales=1001 },  
        new Customer { ID="T", City="Lima", Country="Peru",  
Region="South America", Sales=2002 }  
    };
```

(3) 在Main()方法的customers列表初始化之后，输入（或修改）查询，如下所示：

```
var queryResults = customers.Select(c => c.Region).Dis
```

(4) 完成Main()方法的其余代码，如下所示：

```
foreach (var item in queryResults)  
{  
    WriteLine(item);  
}  
Write("Program finished, press Enter/Return to continue");  
ReadLine();
```

(5) 编译并执行程序，结果显示的是顾客所在的唯一区域，如下所示：

North America

Asia

South America

Europe

Africa

Program finished, press Enter/Return to continue:

示例的说明

`Customer`类和`customers`列表的初始化与前面例子中的相同。在查询语句中，调用了`Select()`方法，用一个简单的Lambda表达式从`Customer`对象中选择区域，再调用`Distinct()`，从`Select()`中返回唯一的结果：

```
var queryResults = customers.Select(c => c.Region).Distinct();
```

只能在方法语法中使用`Distinct()`，所以使用方法语法调用`Select()`。还可以调用`Distinct()`来修改在查询语法中创建的查询：

```
var queryResults = (from c in customers select c.Region).Disti
```

查询语法由C#编译器转换为方法语法中的同系列LINQ方法调用，所以如果可以改进可读性和代码风格，可以混合和匹配它们。

20.10 多级排序

处理了带多个属性的对象后，就要考虑另一种情形了：按一个字段给查询结果排序是不够的，需要查询顾客，并按照区域使结果以字母顺序排列，再按照区域中的国家或城市名称以字母顺序排序。使用LINQ，可以方便地完成这个任务，如下面的示例所示。

试一试：多级排序：

BegVCSharp_20_9_MultiLevelOrdering\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）修改前面的示例BegVCSharp_20_8_SelectDistinctQuery，或在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_9_MultiLevelOrdering。

（2）如BegVCSharp_20_8_SelectDistinctQuery示例所示，创建Customer类并初始化顾客列表（List<Customer>customers），这些代码与前面示例中的代码完全相同。

（3）在Main()方法的customers列表初始化之后，输入如下所示的查询：

```
var queryResults =
```

```
from c in customers
orderby c.Region, c.Country, c.City
select new { c.ID, c.Region, c.Country, c.City }
;
```

(4) 结果处理循环和Main()方法的其余代码与前面例子中的相同。

(5) 编译并执行程序，从所有顾客中选择出来的属性将先按区域排序，再按国家排序，最后按城市排序，如下所示：

```
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = K, Region = North America, Country = Mexico, City = Mex
{ ID = N, Region = North America, Country = USA, City = Los Ar
{ ID = A, Region = North America, Country = USA, City = New Yc
```

```
{ ID = E, Region = South America, Country = Brazil, City = São  
{ ID = S, Region = South America, Country = Colombia, City = E  
{ ID = T, Region = South America, Country = Peru, City = Lima  
Program finished, press Enter/Return to continue:
```

示例的说明

`Customer`类和`customers`列表的初始化与前面例子的相同。因为要查看所有的顾客，这个查询中没有`where`子句，但按顺序列出了要排序的字段，它们放在`orderby`子句的一个用逗号分开的列表中：

```
orderby c.Region, c.Country, c.City
```

这很容易，但不太直观，这个简单的字段列表允许放在`orderby`子句中，但不能放在`select`子句中，不过这就是LINQ的工作方式。如果知道`select`子句会创建一个新对象，而根据定义，`orderby`子句会逐个字段地执行，就不会觉得这个字段列表难以理解了。

可以给列出的任意字段添加`descending`关键字，反转该字段的排序顺序。例如，要对查询结果按照区域升序排序，再按照国家降序排序，只需要在列表中的`Country`后面加上`descending`关键字即可，如下所示：

```
orderby c.Region, c.Country descending, c.City
```

添加了`descending`关键字后，结果如下：

```
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }  
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }  
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }  
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
```

```
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = N, Region = North America, Country = USA, City = Los An
{ ID = A, Region = North America, Country = USA, City = New Yc
{ ID = K, Region = North America, Country = Mexico, City = Mex
{ ID = T, Region = South America, Country = Peru, City = Lima
{ ID = S, Region = South America, Country = Colombia, City = E
{ ID = E, Region = South America, Country = Brazil, City = São
Program finished, press Enter/Return to continue:
```

注意，即使国家的顺序被反转了，印度和中国的城市仍按升序排序。

20.11 组合查询

组合查询（group query）把数据分解为组，允许按组来排序、计算聚合值以及进行比较。这常常是商务环境中最有趣的查询（它驱动了决策系统）。例如，要按照国家或区域比较销售量，确定在哪里开新店或雇佣更多的员工，如下面的示例所示。

试一试：组合查询：

BegVCSharp_20_10_GroupQuery\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_10_GroupQuery。

（2）如BegVCSharp_20_8_SelectDistinctQuery示例所示，创建Customer类并初始化顾客列表（List<Customer>customers），这些代码与前面示例中的代码完全相同。

（3）在Main()方法的customers列表初始化之后，输入如下所示的两个查询：

```
var queryResults =  
    from c in customers
```

```

        group c by c.Region into cg
        select new { TotalSales = cg.Sum(c => c.Sales), Region
;
var orderedResults =
    from cg in queryResults
    orderby cg.TotalSales descending
    select cg
;

```

(4) 在Main()方法中，添加下面的输出语句和foreach处理循环：

```

WriteLine("Total\t: By\nSales\t: Region\n-----\t -----"
foreach (var item in orderedResults)
{
    WriteLine($"{item.TotalSales}\t: {item.Region}");
}

```

(5) 结果处理循环和Main()方法中的其余代码与前面例子中的相同。编译并执行程序，下面是组合结果：

```

Total : By
Sales : Region
-----
52997 : Asia
16999 : North America
12444 : Europe
8558  : South America
7000  : Africa

```


示例的说明

`Customer`类和`customers`列表的初始化与前面例子的相同。

组合查询中的数据通过一个键（`key`）字段来组合，每个组中的所有成员都共享这个字段值。在这个例子中，键字段是`Region`：

```
group c by c.Region
```

要计算每个组的总和，应生成一个新的结果集`cg`：

```
group c by c.Region into cg
```

在`select`子句中，投影了一个新的匿名类型，其属性是总销售量（通过引用`cg`结果集来计算）和组的键值，后者是用特殊的组`Key`来引用的：

```
select new { TotalSales = cg.Sum(c => c.Sales), Region = cg.Key
```

组的结果集实现了LINQ接口`IGrouping`，它支持`Key`属性。我们总是要以某种方式引用`Key`属性，来处理组合的结果，因为该属性表示创建数据中的每个组时使用的条件。

要按`TotalSales`字段对结果降序排序，以便查看哪个区域的销售量最高、哪个区域的销售量次高等等，需要创建第二个查询，对组合查询的结果排序：

```
var orderedResults =  
    from cg in queryResults  
    orderby cg.TotalSales descending
```

```
select cg  
;
```

第二个查询是一个标准的select查询，带一个orderby子句，与前面示例中的相同。但它没有使用任何LINQ组合功能，只是数据源来自于前面的组合查询。

接着输出结果，用一些格式化代码显示带有列标题的数据，在总销售量与组名之间显示了分隔符：

```
WriteLine("Total\t: By\nSales\t: Region\n---\t ---");  
foreach (var item in orderedResults)  
{  
    WriteLine($"{item.TotalSales}\t: {item.Region}");  
};
```

可以用更复杂的方式进行格式化，指定字段宽度，总销售量右对齐，但这只是一个例子，不需要这么多格式，能看清数据，理解代码做了些什么就足够了。

20.12 Join查询

刚才用一个共享的键字段（ID）创建的customers和orders列表等数据集可以执行join查询，即可以用一个查询搜索两个列表中相关的数据，用键字段把结果连接起来。这类似于SQL数据查询语言中的JOIN操作。LINQ在查询语法中提供了join命令，如下面的示例所示。

试一试：Join查询：

BegVCSharp_20_11_JoinQuery\Program.cs

按照下面的步骤在Visual Studio 2015中创建示例：

（1）在C:\BegVCSharp\Chapter20目录中创建一个新的控制台应用程序BegVCSharp_20_11_JoinQuery。

（2）从前面的示例中复制用来创建Customer类和Order类的代码，以及初始化顾客列表（List<Customer>customers）和订单列表（List<Order>orders）的代码。

（3）在Main()方法的customers和orders列表初始化之后，输入如下所示的查询：

```
var queryResults =  
    from c in customers
```

```

join o in orders on c.ID equals o.ID
select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder
            SalesAfter = c.Sales+o.Amount };

```

(4) 用前面例子中使用的标准foreach查询处理循环结束程序：

```

foreach (var item in queryResults)
{
    WriteLine(item);
}

```

(5) 编译并执行程序，结果如下：

```

{ ID = P, City = Tehran, SalesBefore = 7000, NewOrder = 100, S
{ ID = Q, City = London, SalesBefore = 8000, NewOrder = 200, S
{ ID = R, City = Beijing, SalesBefore = 9000, NewOrder = 300,
{ ID = S, City = Bogotá, SalesBefore = 1001, NewOrder = 400, S
{ ID = T, City = Lima, SalesBefore = 2002, NewOrder = 500, Sal
Program finished, press Enter/Return to continue:

```

示例的说明

Customer类和Order类以及customers和orders列表的声明和初始化与前面示例中的相同。

查询使用join关键字通过Customer类和Order类的ID字段，把每个顾客与其对应的订单连接起来：

```

var queryResults =
    from c in customers

```

```
join o in orders on c.ID equals o.ID
```

on关键字之后是键字段（ID）的名称，equals关键字指定另一个集合中的对应字段。查询结果仅包含两个集合中ID字段值相同的对象数据。

select语句投影了一个带指定属性的新数据类型，因此可以清楚地看到最初的总销售量、新订单和最终的新总销售量：

```
select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder = c  
            SalesAfter = c.Sales+o.Amount };
```

这个程序没有在customer对象中递增总销售量，但可以在自己的业务逻辑中完成这一任务。

foreach循环的逻辑和查询中值的显示与本章前面示例中的相同。

20.13 练习

(1) 修改第3个示例程序BegVCSharp_20_3_QuerySyntax，将结果降序排列。

(2) 在大数程序示例BegVCSharp_20_6_LargeNumberQuery中修改传送给GenerateLotsOf-Numbers()方法的数字，创建不同规模的结果集，看看查询结果所受的影响。

(3) 给大数程序示例BegVCSharp_20_6_LargeNumberQuery中的查询添加一个orderby子句，看看这会如何影响性能。

(4) 修改大数程序示例BegVCSharp_20_6_LargeNumberQuery中的查询条件，选择数字列表中的较大和较小子集，看看这会如何影响性能？

(5) 修改方法语法示例BegVCSharp_20_4_MethodSyntax，完全删除where子句，输出量会有多少？

(6) 给第三个示例程序BegVCSharp_20_3_QuerySyntax添加聚合运算符，哪些简单的聚合运算符适用于这种非数字的结果集？

附录A给出了练习答案。

20.14 本章要点

主题	要点
LINQ的概念和使用场合	LINQ是内置于C#中的一种查询语言。使用LINQ可以在大型的对象集合、XML或数据库中查询数据
LINQ查询的组成部分	LINQ查询包含from、where、select和orderby子句
获取LINQ查询的结果的方式	使用foreach语句迭代LINQ查询的结果
延迟执行	LINQ查询会延迟到执行foreach语句时执行
方法语法和查询语法	简单的LINQ查询使用查询语法，较高级的查询使用方法查询。对于任意给定的查询，查询语法和方法语法的结果相同
Lambda表达式	Lambda表达式可以使用方法语法，随时声明一个用于LINQ查询的方法
聚合运算符	使用LINQ聚合运算符获得大型数据集的信息，而不必迭代每个结果
组合查询	使用组合查询给数据分组，再按照组进行排序、计算聚合值以及进行比较
排序	使用orderby运算符给查询的结果排序
join运算符	使用join运算符在一个查询中查找多个集合中的相关数

据

第21章 数据库

本章内容：

- 使用数据库
- 理解Entity Framework
- 用Code First创建数据
- 使用LINQ和数据库
- 导航数据库关系
- 在数据库中创建和查询XML

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡中下载Chapter 21 Code后，可以找到与本章示例对应的单独文件。

上一章介绍了LINQ（Language-Integrated Query），展示了LINQ如何使用对象和XML。本章将学习如何将对象存储在数据库中，并使用LINQ查询数据。

21.1 使用数据库

数据库是永久的、结构化的数据仓库。有许多不同种类的数据库，但存储和查询业务数据的最常见类型是关系数据库，如Microsoft SQL Server和Oracle。关系数据库使用SQL数据库语言（SQL代表结构化查询语言，Structured Query Language）来查询操纵它们的数据。传统上，使用这样的数据库至少需要知道一些SQL知识，以便在编程语言中嵌入的SQL语句，或在面向SQL的数据库类库中把包含SQL语句的字符串传递给API调用或方法。

听起来很复杂，不是吗？好消息是，有了Visual C# 2015，可以使用Code First方法在C#中创建对象，存储在数据库中，并使用LINQ查询对象，而不必使用另一种语言，比如SQL。

21.2 安装SQL Server Express

要运行本章中的示例，必须安装Microsoft SQL Server Express，这是Microsoft SQL Server的免费轻量级版本。我们将使用LocalDB选项与SQL Server Express，以允许Visual Studio 2015直接创建和打开数据库文件，而不必连接到单独的服务器上。

带有LocalDB的SQL Server Express支持的SQL语法与完整的Microsoft SQL Server相同，所以它是适合于初学者的版本。下载SQL Server Express的链接如下：

<http://www.microsoft.com/en-us/server-cloud/products/sql-server-editions/sqlserver-express.aspx>

注意：如果熟悉SQL Server，拥有Microsoft SQL Server实例的访问权限，就可以跳过这个安装步骤，但需要改变连接信息，以匹配自己的SQL Server实例。如果从未使用过SQL Server，就安装SQL Server Express。

21.3 Entity Framework

.NET中支持Code First的类库是Entity Framework的最新版本。这个名字来源于一个数据库概念：实体关系模型。其中实体是数据对象（如客户）的抽象概念，它与关系数据库中的其他实体（如订单和产品）相关，例如客户订下了某产品。

Entity Framework将C#程序中的对象映射到关系数据库的实体上。这就是所谓的对象-关系映射。对象-关系映射是将C#中的类、对象和属性映射到构成关系数据库的表、行和列的代码。手工创建这个映射代码非常繁杂、耗时，但Entity Framework使它很容易完成。

Entity Framework建立在ADO.NET的基础上，而ADO.NET是基于.NET的低层数据访问库。ADO.NET需要一些SQL的知识，但幸运的是，Entity Framework已经自动处理了这个问题，用户可以专注于C#代码。

注意：在技术上，Entity Framework的全称是ADO.NET Entity Framework。在Visual Studio的一些地方会通过全名引用它。另一方面，在许多博客和文章中，Entity Framework缩写为EF。

Entity Framework还带有LINQ to Entities，这是Entity Framework的LINQ提供程序，使用它很便于在C#中查询数据库。下面就开始在数据

库中创建一些对象。

21.4 Code First数据库

下面的例子使用Code First和Entity Framework在数据库中创建一些对象，然后使用LINQ to Entities查询创建的对象。

试一试：Code

BegVCSharp_21_1_CodeFirstDatabase

First数据库：

按照以下步骤在Visual Studio 2015中创建例子：

(1) 在目录C:\BegVCSharp\Chapter21下创建一个新的控制台应用程序项目BegVCSharp_21_1_CodeFirstDatabase。

(2) 单击OK以创建项目。

(3) 为了添加Entity Framework，按第19章的方式使用NuGet。选择Tools|NuGet Package Manager|Manage NuGet Packages for Solution，如图21-1所示。

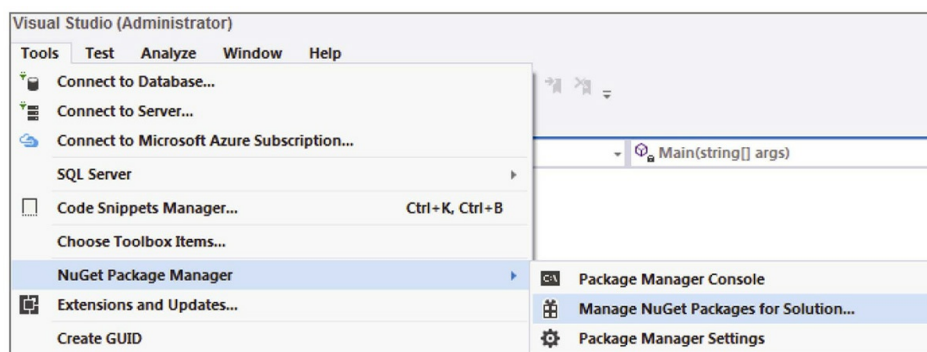


图21-1

（4）取消Include Prerelease复选框的选择，获得Entity Framework的最新稳定版本，如图21-2所示。点击Install按钮。

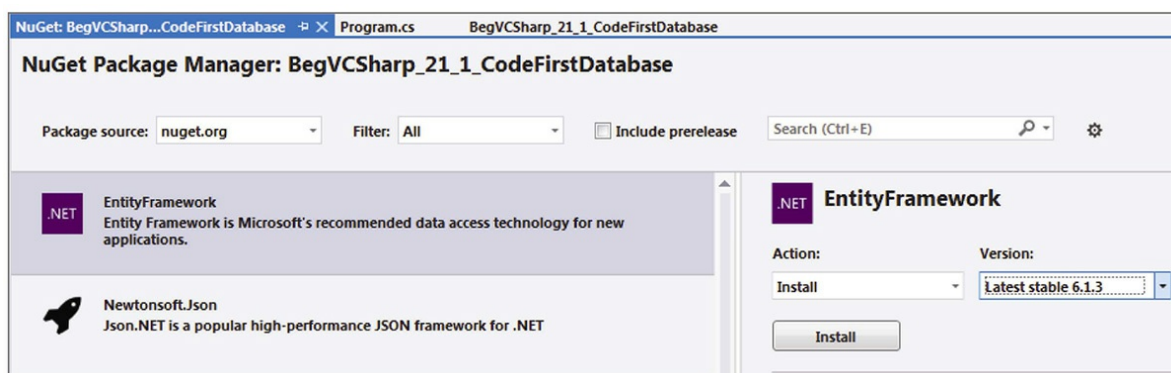


图21-2

（5）在Preview对话框中单击OK，如图21-3所示。

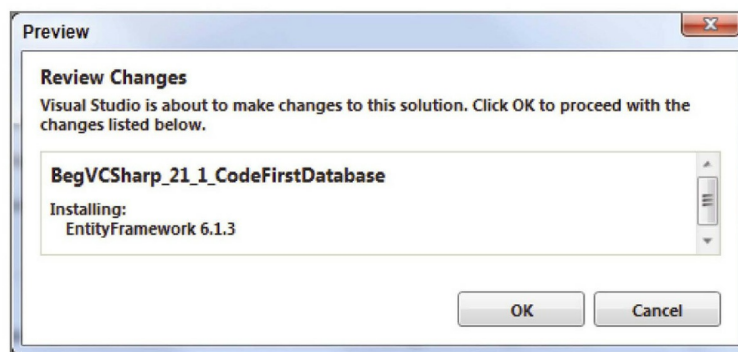


图21-3

（6）现在Entity Framework的License Acceptance对话框如图21-4所示。单击I Accept按钮。

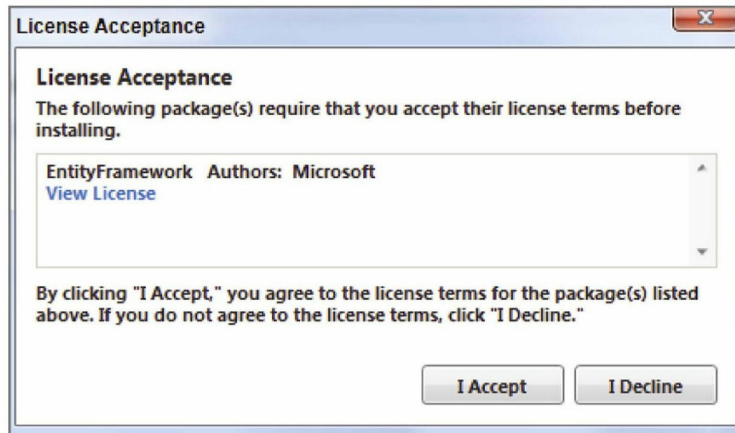


图21-4

(7) 现在，Entity Framework及其引用已添加到项目中。在Solution Explorer的References部分可以看到它们。如图21-5所示。

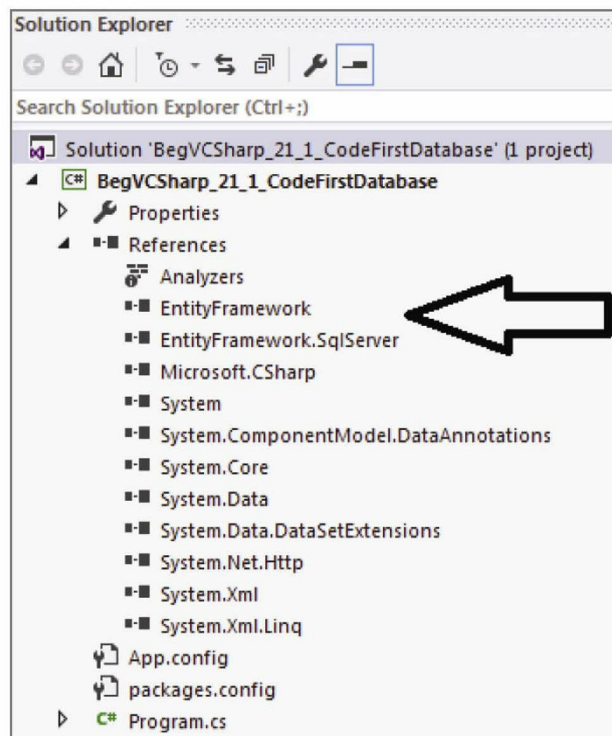


图21-5

(8) 打开Program.cs主源文件，并添加以下代码。首先在文件顶部、其他using子句的下面添加Entity Framework名称空间：

```
using System.Data.Entity;
```

(9) 接下来，给数据注解添加另一个using子句，以便给Entity Framework提示如何建立数据库。最后，与之前的例子相同，添加System.Console名称空间：

```
using System.ComponentModel.DataAnnotations;
```

```
using static System.Console;
```

(10) 接下来添加一个Book类，其中的Author、Title和Code类似于第19章使用的例子。Code字段前的[Key]特性是一个数据注解，告诉C#使用这个字段作为数据库中每个对象的唯一标识符。

```
namespace BegVCSharp_21_1_CodeFirstDatabase
```

```
{
```

```
public class Book
```

```
{
```

```
public string Title { get; set; }
```

```
public string Author { get; set; }
```

```
[Key] public int Code { get; set; }
```

```
}
```

（11）现在添加DbContext类（数据库上下文），来管理创建、更新和删除数据库中的书籍表：

```
public class BookContext : DbContext
```

{

```
public DbSet<Book> Books { get; set; }
```

}

（12）接下来，在Main()函数中添加代码，创建两个Book对象，并保存到数据库中：

```
class Program
```

 $\{$

[illegible]

```
db.Books.Add(book2);
```

```
db.SaveChanges();
```

（13）最后，为一个简单LINQ查询添加代码，列出创建后数据库中的书籍：

```
var query = from b in db.Books
```

```
orderby b.Title
```

```
select b;
```

```
WriteLine("All books in the database:");
```

```
foreach (var b in query)
```

```
{
```

```
    WriteLine($"{b.Title} by {b.Author}, code=
```

```
}
```

```
WriteLine("Press a key to exit...");
```

```
ReadKey();
```

```
}
```

程序的完整代码现在如下：

```
using System.Data.Entity;
using System.Data.Annotations;
using static System.Console;
namespace BegVCSsharp_21_1_CodeFirstDatabase
{
    public class Book
    {
        public string Title { get; set; }
        public string Author { get; set; }
        public int Code { get; set; }
    }
    public class BookContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BookContext())
            {
                Book book1 = new Book { Title = "Beginning Visual C#
```

```

        Author = "Perkins, Reid, and Hammer" };
db.Books.Add(book1);
Book book2 = new Book { Title = "Beginning XML",
        Author = "Fawcett, Quin, and Ayers"};
db.Books.Add(book2);
db.SaveChanges();
var query = from b in db.Books
            orderby b.Title
            select b;
WriteLine("All books in the database:");
foreach (var b in query)
{
    WriteLine($"{b.Title} by {b.Author}, code={b.Code}");
}
WriteLine("Press a key to exit...");
ReadKey();
    }
}
}
}

```

(14) 编译并执行程序（可以按F5开始调试）。书籍数据库的信息如图21-6所示。

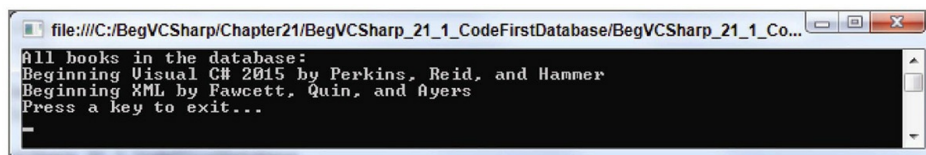


图21-6

按任意键结束程序，关闭控制台屏幕。如果使用Ctrl+F5（开始但不调试），就可能需要按回车键两次。这会结束程序的运行。现在看看它是如何工作的。

示例的说明

如前一章所示，这段代码使用了System.Linq名称空间中的扩展类，创建项目时，Visual C# 2015会自动插入一个using语句，来引用该名称空间：

```
using System.Linq;
```

接下来在文件顶部其他using子句的后面添加Entity Framework名称空间：

```
using System.Data.Entity;
```

然后为数据注解添加using子句，以便添加提示，告诉Entity Framework如何建立数据库，再添加静态的System.Console名称空间：

```
using System.ComponentModel.DataAnnotations;  
using static System.Console;
```

接下来添加了一个Book类，其Author、Title和Code类似于第19章中使用的例子。使用[Key]特性把Code属性识别为数据库中每一行的唯一标识符。

```
namespace BegVCSharp_21_1_CodeFirstDatabase
{
    public class Book

    {

        public string Title { get; set; }

        public string Author { get; set; }

        [Key] public int Code { get; set; }

    }
}
```

之后创建BookContext类，它继承了Entity Framework中的DbContext（数据库上下文）类，用于在需要时创建、更新和删除数据库中的book对象：

```
public class BookContext : DbContext

{

    public DbSet<Book> Books { get; set; }

}
```

类成员DbSet<Book>是一个包含数据库中所有Book实体的集合。

接下来添加代码，以使用BookContext创建两个Book对象，保存到数据库中：

```
using (var db = new BookContext())
{
```

```
Book book1 = new Book { Title = "Beginning Visual C#  
    Author = "Perkins, Reid, and Hammer" };  
db.Books.Add(book1);  
Book book2 = new Book { Title = "Beginning XML",  
    Author = "Fawcett, Quin, and Ayers"};  
db.Books.Add(book2);  
db.SaveChanges();
```

using (var db=new BookContext())子句允许创建一个新的BookContext实例，用于花括号中的所有后续代码。除了方便速记之外，using()子句还确保结束程序时，即使有异常或其他意外的事件，数据库连接和其他与连接相关的底层对象会正确关闭。

Book创建和赋值语句，例如：

```
Book book = new Book { Title = "Beginning Visual C# 2  
    Author = "Perkins, Reid, and Hammer" };
```

是相当简单的Book对象创建语句；没有出现什么数据库魔法。因为这些都是在内存中的简单对象。注意，没有给Code属性分配任何值；目前，未赋值的Code属性只包含一个默认值。

接下来将对BookContext db的更改保存到数据库中：

```
db.SaveChanges();
```

现在出现了一些奇怪的事情，因为使用[Key]特性把Code识别为一个键，把每个对象保存到数据库中时，将一个唯一的值分配给Code字段。不需要使用这个值，甚至不需要在乎它是什么，因为Entity

Framework会自动处理它。

注意： 如果没有把[Key]特性添加到对象中，程序运行时，就会显示一个如图21-7所示的异常。

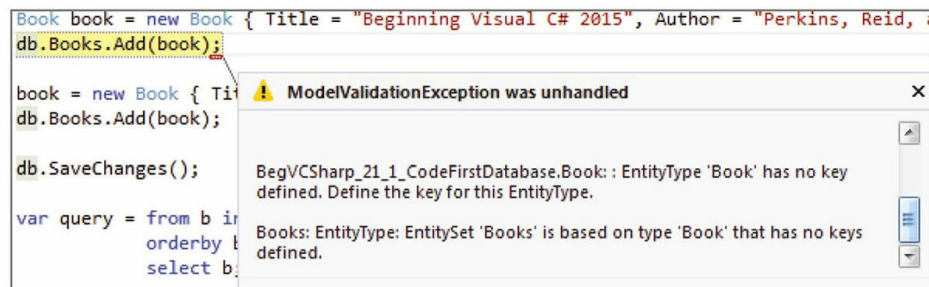


图21-7

最后，给一个简单的LINQ查询执行代码，列出创建后数据库中的书籍：

```
var query = from b in db.Books
            orderby b.Title
            select b;
WriteLine("All books in the database:");
foreach (var b in query)
{
    WriteLine($"{b.Title} by {b.Author}, code={b.Code}");
}
WriteLine("Press a key to exit...");
ReadKey();
```

```
}
```

这个LINQ查询非常类似于前一章使用的查询，但它不使用LINQ to Objects提供程序查询内存中的对象，而是用LINQ to Entities提供程序查询数据库。LINQ根据查询中引用的类型推断正确的提供程序；不需要对逻辑进行任何修改。

最后在退出前，只使用标准的ReadKey()，来暂停程序，以便看到输出。

这很容易，对吧？创建一些对象，保存到数据库中，使用LINQ查询数据库。

21.5 数据库的位置

创建的数据库位于哪个位置？我们永远不会指定文件名或文件夹位置——好奇怪！在Visual Studio 2015中通过Server Explorer可以看到它。进入Tools|Connect to Database。Entity Framework将创建一个数据库，放在它在计算机上找到的第一个本地SQL Server实例中。

如果计算机里以前从来没有任何数据库，Visual C# 2015就会自动创建一个本地SQL Server实例（localdb）\MSSQLLocalDB。要连接到该数据库，在Server Name字段中输入（localdb）\MSSQL-LocalDB，如图21-8所示。

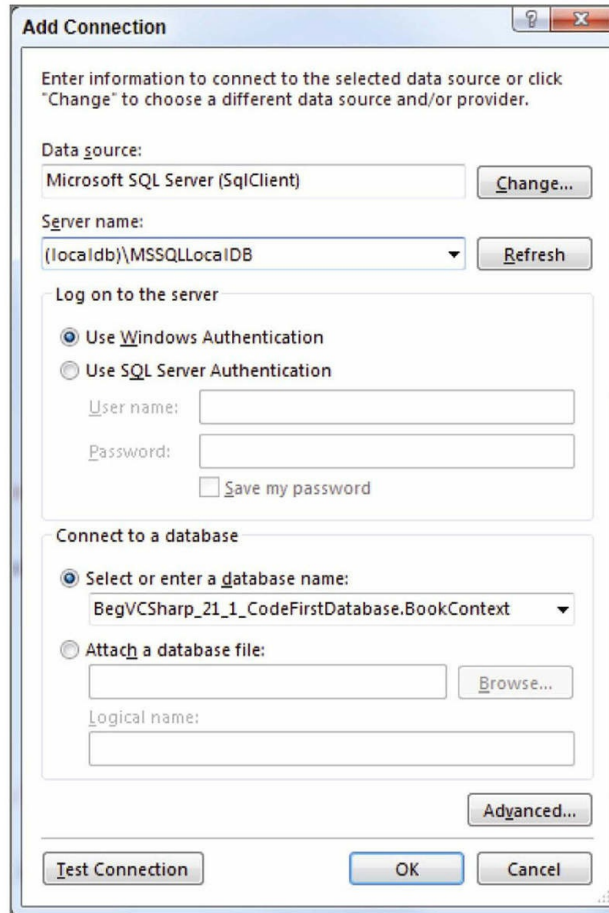


图21-8

注意： 如果使用Visual C# 2015前安装了Visual Studio的一个以前版本，就可能需要在Server Name字段中输入（localdb）\v11.0，因为这是前一版的本地数据库名。如果安装了SQL Server Express Edition，就可能需要输入.\sqlexpress，因为Entity Framework会使用它找到的第一个本地SQL Server数据库。

假设在例子中输入的名字与本章所示完全相同，则包含数据的数据库称为BegVCSsharp_21_1_CodeFirstDatabase.BookContext。花点时间连

接后，它会出现现在Select or enter a database name字段中。

现在可以按下OK，数据库将出现在Visual C# 2015的Server Explorer Data Connections窗口中，如图21-9所示。

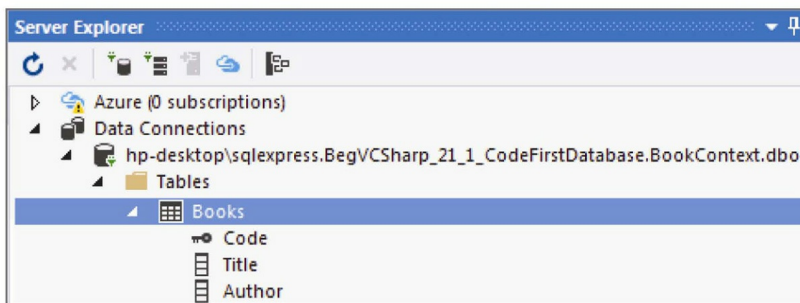


图21-9

现在就可以直接探索数据库。例如，可以右击Books表，并选择Show Table Data，看到自己输入的数据，如图21-10所示。

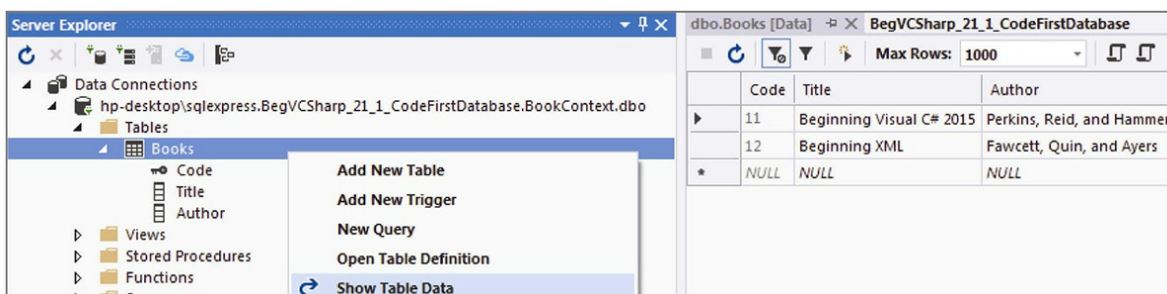


图21-10

21.6 导航数据库关系

Entity Framework最强大的一个方面是它能够自动创建LINQ对象，帮助找到数据库中相关的表之间的关系。

下面的示例将添加两个与Book类相关的新类，生成一个简单的书店库存报告。新类称为Store（代表每个书店）和Stock（代表在商店货架上的书和从出版商那里订购的书）。这些新类和关系的图如图21-11所示。

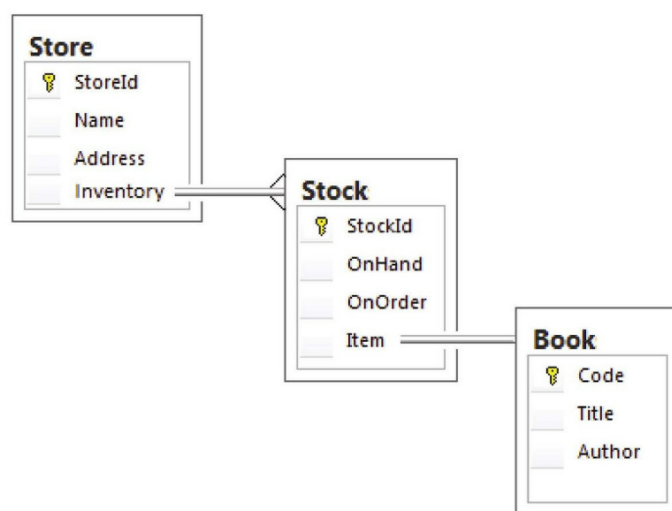


图21-11

每个商店都有名称、地址和库存集合（由一个或多个Stock对象组成，每个Stock对象对应书店中每本不同的书（书名）。Store和Stock之间是一对多的关系。每个Stock记录正好与一本书有关。Stock和Book之间是一对一的关系。需要库存记录，因为一个商店可能有三本相同的书，但另一个商店可能有六本相同的书。

有了Code First，就只需要创建C#对象和集合，而Entity Framework会自动创建数据库结构，以便你轻松地导航数据库对象之间的关系，然后在数据库中查询相关对象。

试一试：导航数据库关系：

BegVCSharp_21_2_DatabaseRelations

按照以下步骤在Visual Studio 2015中创建例子：

(1) 在目录C:\BegVCSharp\Chapter21下创建一个新的控制台应用程序项目BegVCSharp_21_2_DatabaseRelations。

(2) 单击OK以创建项目。

(3) 使用NuGet添加Entity Framework，与前面的例子一样。选择Tools|NuGet Package Manager|Manage NuGet Packages for Solution。

(4) 在NuGet Package Manager中，选择Entity Framework，取消选中Include Prerelease复选框，获得Entity Framework最新的稳定版本。单击Install按钮。它不需要下载，因为前一步已经下载了它。单击Preview Changes上的OK，单击License Acceptance对话框中的I Accept按钮。

(5) 打开Program.cs主源文件。与前面的示例一样，给System.Console、System.Data.Entity和DataAnnotations名称空间添加using语句，以及创建Book类的代码：

```
using System.Data.Entity;
```

```

using System.ComponentModel.DataAnnotations;
using static System.Console;
namespace BegVCSharp_21_2_DatabaseRelations
{
    public class Book
    {
        public string Title { get; set; }
        public string Author { get; set; }
        [Key]
        public int Code { get; set; }
    }
}

```

（6）现在声明Store和Stock类，如下所示。确保Inventory和Item声明为virtual。后面会说明其原因。

```

public class Store

```

```

{

```

```

    [Key]

```

```
public int StoreId { get; set; }
```

```
public string Name { get; set; }
```

```
public string Address { get; set; }
```

```
public virtual List<Stock> Inventory { get; set; }
```

```
}
```

```
public class Stock
```

```
{
```

[Key]

```
public int StockId { get; set; }
```

```
public int OnHand { get; set; }
```

```
public int OnOrder { get; set; }
```

```
public virtual Book Item{ get; set; }
```

```
}
```

(7) 接着给DbContext类添加Stores和Stocks:

```
public class BookContext : DbContext  
{
```

```
public DbSet<Book> Books { get; set; }
```

```
public DbSet<Store> Stores { get; set; }
```

```
public DbSet<Stock> Stocks { get; set; }
```

```
}
```

(8) 现在将代码添加到Main()方法中，以使用BookContext，创建Book类的两个实例，与前面的例子一样：

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new BookContext())
        {
            Book book1 = new Book
            {
                Title = "Beginning Visual C# 2015",
                Author = "Perkins, Reid, and Hammer"
```

```

};
db.Books.Add(book1);
Book book2 = new Book
{
    Title = "Beginning XML",
    Author = "Fawcett, Quin, and Ayers"
};
db.Books.Add(book2);
}

```

(9) 现在，在using (var db=newBookContext())子句中给第一个书店添加实例及其库存：

```

var store1 = new Store

```

```

{

```

```

    Name = "Main St Books",

```

```

    Address = "123 Main St",

```



```
Inventory = new List<Stock>()
```

```
};
```

```
db.Stores.Add(store1);
```

```
Stock store1book1 = new Stock
```

```
{ Item = book1, OnHand = 4, OnOrder = 6 };
```

```
store1.Inventory.Add(store1book1);
```

```
Stock store1book2 = new Stock
```

```
{ Item = book2, OnHand = 1, OnOrder = 9 };
```

```
store1.Inventory.Add(store1book2);
```

(10) 给第二个书店添加实例及其库存:

```
var store2 = new Store
```

```
{
```

```
    Name = "Campus Books",
```

```
    Address = "321 College Ave",
```

```
Inventory = new List<Stock>()
```

```
};
```

```
db.Stores.Add(store2);
```

```
Stock store2book1 = new Stock
```

```
{ Item = book1, OnHand = 7, OnOrder = 23 };
```

```
store2.Inventory.Add(store2book1);
```

```
Stock store2book2 = new Stock
```

```
{ Item = book2, OnHand = 2, OnOrder = 8 };
```

```
store2.Inventory.Add(store2book2);
```

(11) 接着保存数据库的修改，与前面的例子相同：

```
db.SaveChanges();
```

(12) 现在在所有的书店上创建一个LINQ查询，并输出结果：

```
var query = from store in db.Stores
```

```
orderby store.Name
```

```
select store;
```

(13) 最后添加代码，输出查询的结果，并暂停：

```
WriteLine("Bookstore Inventory Report:");
```

```
foreach (var store in query)
```

```
{
```

```
WriteLine($"{store.Name} located at {store.Address}");
```

```
foreach (Stock stock in store.Inventory)
```

```
{
```

```
    WriteLine($"- Title: {stock.Item.Title}");
```

```
    WriteLine($"-- Copies in Store: {stock.OnHand}");
```

```
    WriteLine($"-- Copies on Order: {stock.OnOrder}")
```

```
}
```

```
}
```

```
WriteLine("Press a key to exit...");
```

```

        ReadKey();
    }
}
}
}

```

(14) 编译并执行程序（可以按F5，开始调试）。书店库存的信息如图21-12所示。

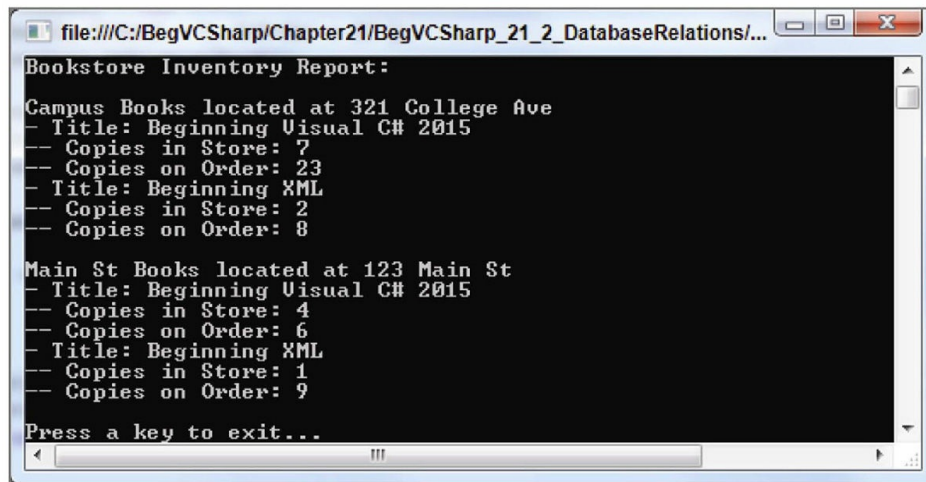


图21-12

按任意键结束程序，关闭控制台屏幕。如果使用Ctrl+F5（开始但不调试），就可能需要按回车键两次。结束程序的运行。现在看看它是如何工作的。

示例的说明

Entity Framework的基础DbContext和数据注解在前面的例子中讲过了。所以这里只关注不同的地方。

Store和Stock类类似于原来的Book类，但给Inventory和Item添加了一些新的virtual属性，如下所示：

```
public class Store
```

```
{
```

```
[Key]
```

```
public int StoreId { get; set; }
```

```
public string Name { get; set; }
```

```
public string Address { get; set; }
```



```
public virtual List<Stock> Inventory { get; set; }  
  
}
```

```
public class Stock
```

```
{
```

```
[Key]
```

```
public int StockId { get; set; }
```

```
public int OnHand { get; set; }
```

```
public int OnOrder { get; set; }
```

```
public virtual Book Item{ get; set; }
```

```
}
```

Inventory属性的外观和行为像一个普通的内存中List<Stock>集合。然而，因为它声明为virtual，所以在数据库中存储和检索它时，Entity Framework可以重写其行为。

Entity Framework负责数据库的详细信息，比如给数据库中的Stocks表添加一个外键列，实现Store和Stock记录之间的Inventory关系。同样，Entity Framework将另一个外键列添加到数据库的Stock表中，实现Stock和Book之间的Item关系。在BegVCSharp_21_2_DatabaseRelations.BookContext数据库的Server Explorer数据库设计视图中可以看到它们，如图21-13所示。

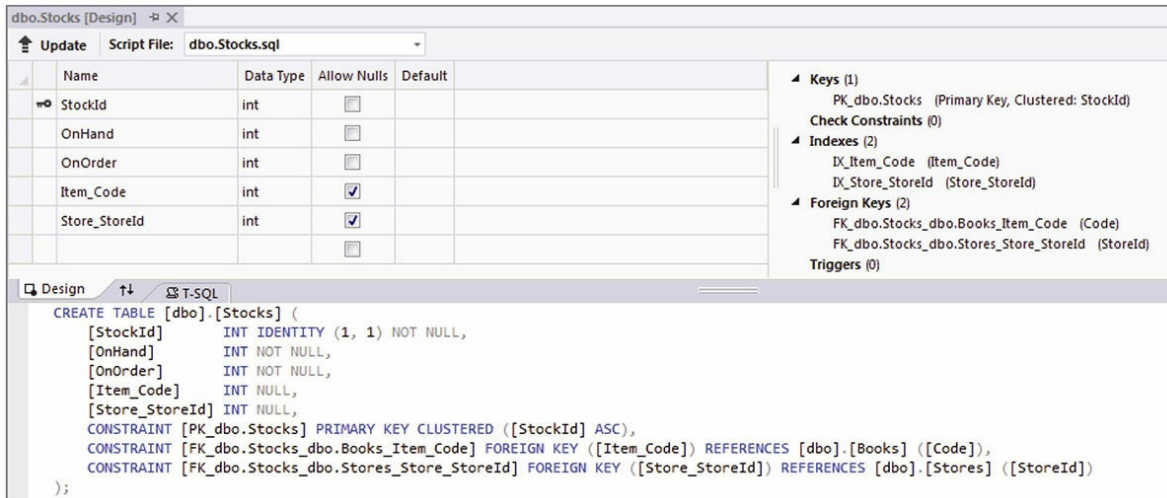


图21-13

过去，必须决定如何将程序中的集合映射到数据库中的外键和列上，并在设计变化时，使这些代码保持最新。然而，有了Entity Framework，就不需要知道这些细节。有了Code First，只需要处理C#类和集合，Entity Framework会自动完成其他工作。

接下来在BookContext中添加Store和Stock的DbSet类。

```

public class BookContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Store> Stores { get; set; }

    public DbSet<Stock> Stocks { get; set; }
}

```

```
}
```

然后用这些DbSet类来创建两本书、两个商店和两个库存记录（每个商店中的每本书）的实例：

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new BookContext())
        {
            Book book1 = new Book
            {
                Title = "Beginning Visual C# 2015",
                Author = "Perkins, Reid, and Hammer"
            };
            db.Books.Add(book1);
            Book book2 = new Book
            {
                Title = "Beginning XML",
                Author = "Fawcett, Quin, and Ayers"
            };
            db.Books.Add(book2);
            var store1 = new Store
```

```
{
```

```
    Name = "Main St Books",
```

```
    Address = "123 Main St",
```

```
    Inventory = new List<Stock>()
```

```
};
```

```
db.Stores.Add(store1);
```

```
Stock store1book1 = new Stock
```

```
{ Item = book1, OnHand = 4, OnOrder = 6 };
```

```
store1.Inventory.Add(store1book1);
```

```
Stock store1book2 = new Stock
```

```
{ Item = book2, OnHand = 1, OnOrder = 9 };
```

```
store1.Inventory.Add(store1book2);
```

```
var store2 = new Store
```

```
{
```

```
Name = "Campus Books",
```

```
Address = "321 College Ave",
```

```
Inventory = new List<Stock>()
```

```
};
```

```
db.Stores.Add(store2);
```

```
Stock store2book1 = new Stock
```

```
{ Item = book1, OnHand = 7, OnOrder = 23 };
```

```
store2.Inventory.Add(store2book1);
```

```
Stock store2book2 = new Stock
```

```
{ Item = book2, OnHand = 2, OnOrder = 8 };
```

```
store2.Inventory.Add(store2book2);
```

创建对象后，就把更改保存到数据库中：

```
db.SaveChanges();
```


然后建立一个简单的LINQ查询，列出所有商店的信息：

```
var query = from store in db.Stores
```

```
    orderby store.Name
```

```
    select store;
```

打印查询结果的代码非常简单，因为它只处理对象和集合，没有数据库的特定代码：

```
WriteLine("Bookstore Inventory Report:");
```

```
foreach (var store in query)
```

```
{
```

```
WriteLine($"{store.Name} located at {store.Address}"
```

```
foreach (Stock stock in store.Inventory)
```

```
{
```

```
    WriteLine($"- Title: {stock.Item.Title}");
```

```
    WriteLine($"-- Copies in Store: {stock.OnHand}");
```

```
    WriteLine($"-- Copies on Order: {stock.OnOrder}");
```

```
}
```

```
}
```

为了打印每个商店的库存，只需使用一个foreach循环，与处理任何集合一样。

21.7 处理迁移

开发代码时，难免有改变想法的时候。属性可能有更好的名称，或者发现需要一个新的类或关系。如果通过Entity Framework改变类中连接到数据库的代码，第一次运行改变了的程序时，就会遇到如图21-14所示的Invalid Operation Exception。

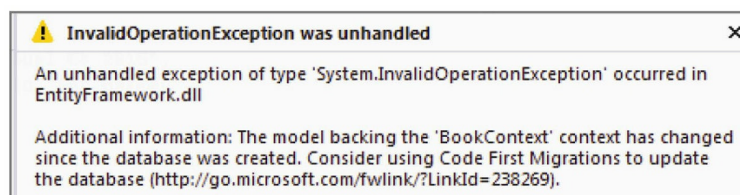


图21-14

使数据库与改变的类保持一致是很复杂的，但有了Entity Framework，步骤会相对容易。错误消息显示，需要向程序添加Code First Migrations包。

为此，进入Tools|NuGet Package Manager|Package Manager Console。这将打开一个命令窗口，如图21-15所示。

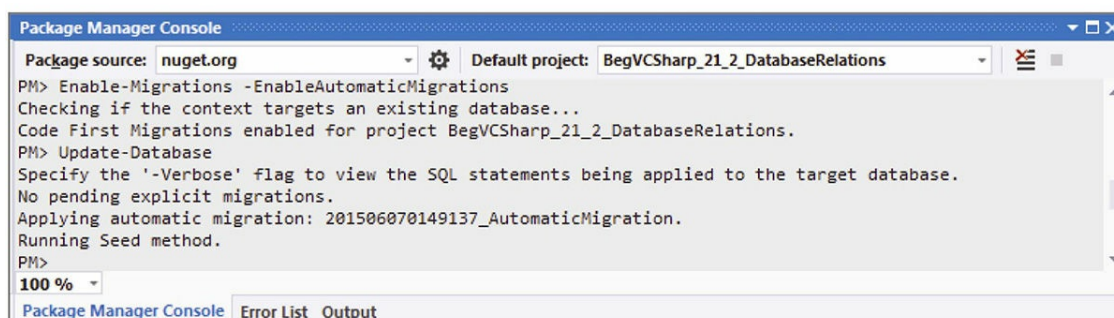


图21-15

要启用把数据库自动迁移到更新的类结构中，应在Package Manager Console的PM>提示下输入如下命令：

```
Enable-Migrations -EnableAutomaticMigrations
```

这会把Migrations类添加到项目中，如图21-16所示。

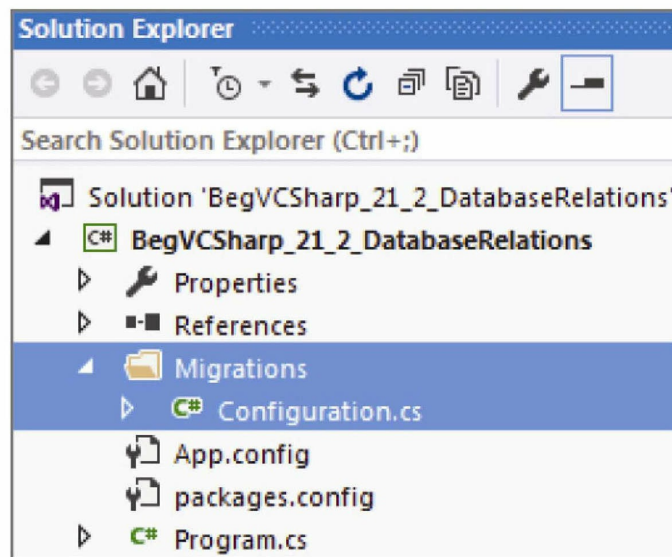


图21-16

Entity Framework会比较数据库和程序的时间戳，当数据库与类不同步时，建议同步。要更新数据库，只需要在Package Manager Console的PM>提示下输入如下命令：

```
Update-Database
```

21.8 在已有的数据库中创建和查询XML

最后一个例子将结合前面所学的LINQ、数据库和XML。

XML通常用于在客户机和服务器之间的数据通信或多层应用程序的“层”之间的数据通信。我们常常要在数据库中查询一些数据，然后从该数据中生成一个XML文档或片段，传递到另一层。

下面的示例将创建一个查询，在前面的示例数据库中查找一些数据，使用LINQ to Entities查询数据，然后使用LINQ to XML类把数据转换为XML。这是一个Database First示例，而不是Code First编程例子，它利用现有的数据库，并从中生成C#对象。

试一试：从数据库中生成XML：

BegVCSharp_21_3_XMLfromDatabase

按照以下步骤在Visual Studio 2015中创建例子：

(1) 在目录C:\BegVCSharp\Chapter21下创建一个新的控制台应用程序BegVCSharp_21_3_XMLfromDatabase。

(2) 如前面的示例所述，把Entity Framework添加到项目中。

(3) 给前面示例中使用的数据库添加连接，方法是选择 Project|Add New Item，在Add New Item对话框中选择ADO.NET Entity Data Model，把名字从Model1改为BookContext，如图21-17所示。

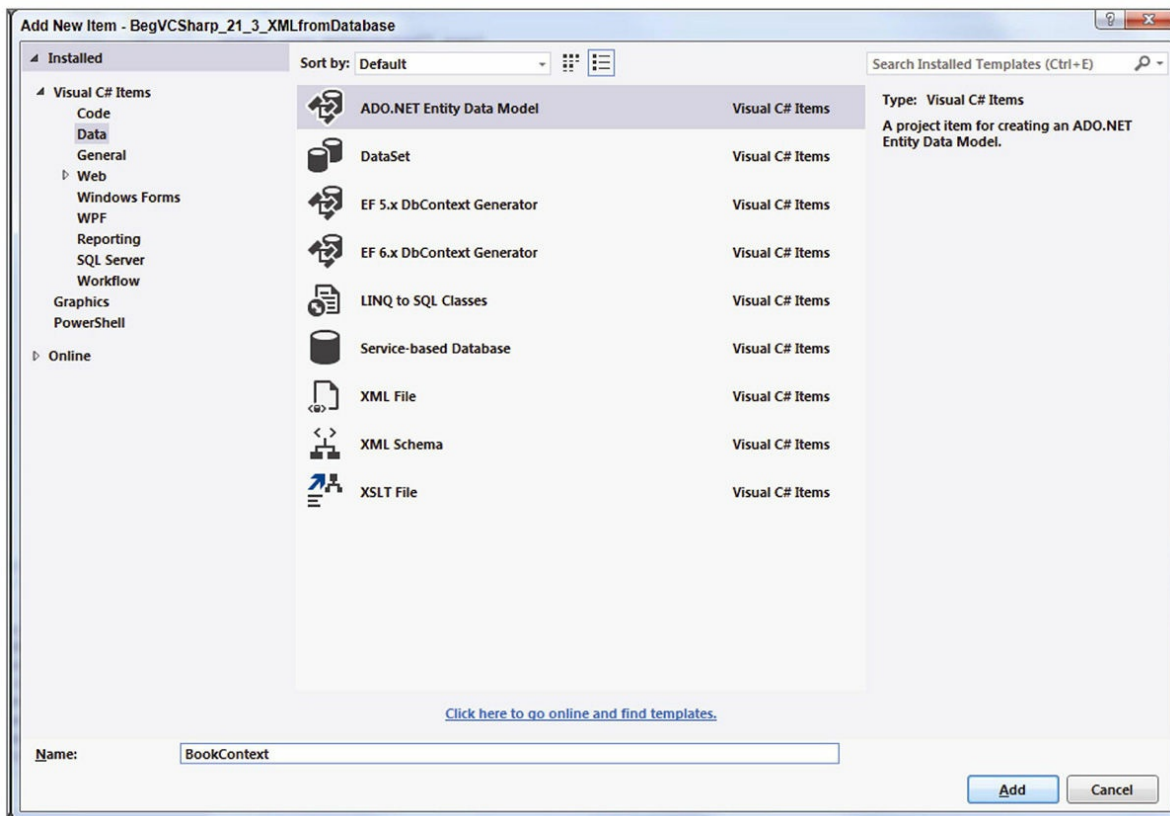


图21-17

(4) 在Entity Data Model Wizard中，选择到前面示例创建的 BegVCSharp_21_2_DatabaseRelations. BookContext数据库的连接，如图 21-18所示。

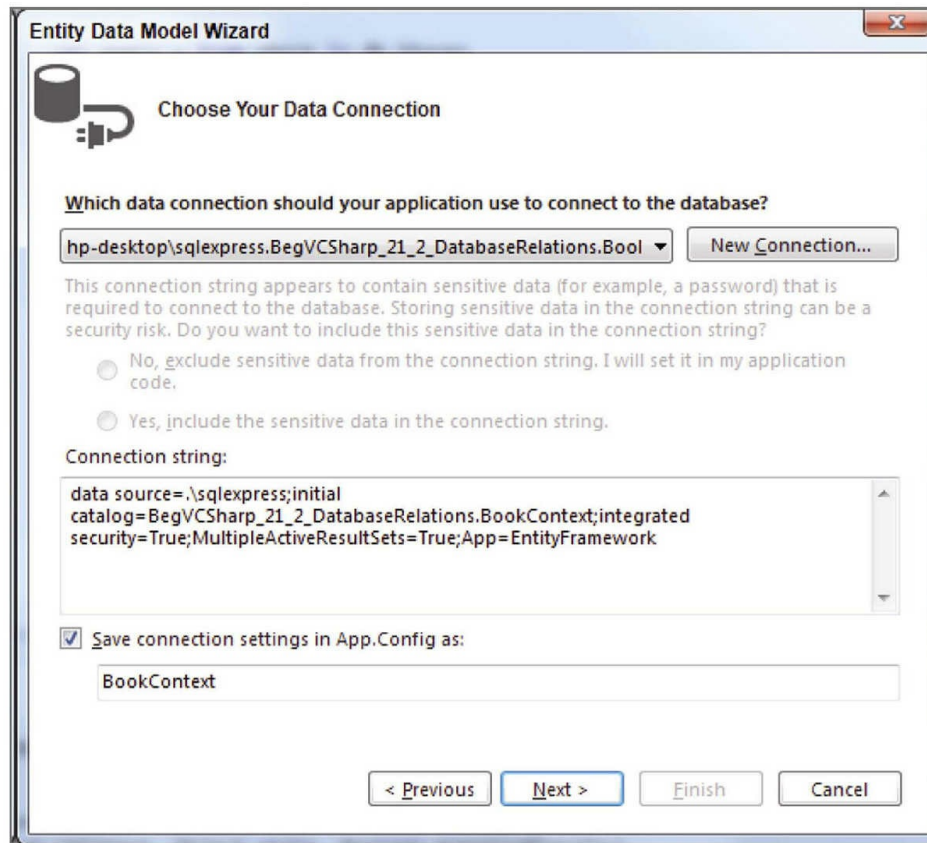


图21-18

(5) 打开Program.cs主源文件。

(6) 在Program.cs的开头添加对System.Xml.Linq名称空间的一个引用，如下所示：

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Xml.Linq;
```



```
foreach (var s in query)
```

```
{
```

```
    XElement storeElement = new XElement("store",
```

```
        new XAttribute("name", s.Name),
```

```
        new XAttribute("address", s.Address),
```

```
        from stock in s.Stocks
```

```
        select new XElement("stock",
```

```
new XAttribute("StockID", stock.StockId),
```

```
new XAttribute("onHand",
```

```
stock.OnHand),
```

```
new XAttribute("onOrder",
```

```
stock.OnOrder),
```

```
new XElement("book",
```

```
new XAttribute("title",
```

```
stock.Book.Title),
```

```
new XAttribute("author",
```

```
stock.Book.Author)
```

```
)// end book
```

```
) // end stock
```

```
); // end store
```

```
WriteLine(storeElement);
```

```
}
```

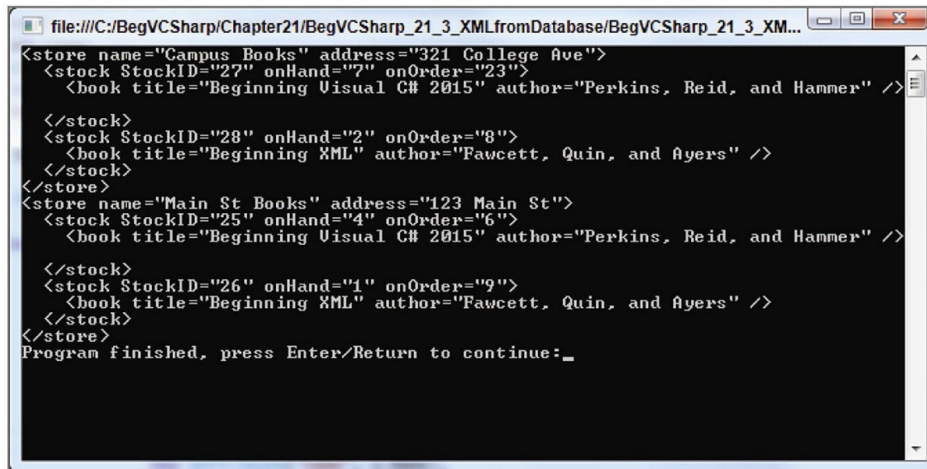
```
Write("Program finished, press Enter/Return to continue:");
```

```
ReadLine();
```

```
}
```

```
}
```

(8) 编译并执行程序（可以按F5，开始调试）。输出如图21-19所示。



```
file:///C:/BegVCSharp/Chapter21/BegVCSharp_21_3_XMLfromDatabase/BegVCSharp_21_3_XM...
<store name="Campus Books" address="321 College Ave">
  <stock StockID="27" onHand="7" onOrder="23">
    <book title="Beginning Visual C# 2015" author="Perkins, Reid, and Hammer" />
  </stock>
  <stock StockID="28" onHand="2" onOrder="8">
    <book title="Beginning XML" author="Fawcett, Quin, and Ayers" />
  </stock>
</store>
<store name="Main St Books" address="123 Main St">
  <stock StockID="25" onHand="4" onOrder="6">
    <book title="Beginning Visual C# 2015" author="Perkins, Reid, and Hammer" />
  </stock>
  <stock StockID="26" onHand="1" onOrder="9">
    <book title="Beginning XML" author="Fawcett, Quin, and Ayers" />
  </stock>
</store>
Program finished, press Enter/Return to continue:_
```

图21-19

按回车键退出程序，关闭控制台屏幕。如果使用Ctrl+F5（开始但不调试），就可能需要按回车键两次。

示例的说明

在Program.cs中添加了对System.Xml.Linq名称空间的引用，以调用LINQ to XML构造函数类和Entity Framework类。

在Add New Item对话框中选择ADO.NET Entity Data Model，添加Database First代码时，Visual Studio会使用前面例子创建的现有数据库BegVCSharp_21_2_DatabaseRelations.BookContext的信息，生成一个单独的BookContext.cs类，并添加到项目中。

在主程序中，创建了BooksContext数据库上下文类的实例和与前面例子相同的LINQ to Entities查询：

```
using (var db = new BookContext())
```

```
{
```

```
var query = from store in db.Stores
```

```
orderby store.Name
```

```
select store;
```

在foreach循环里处理查询的结果时，使用LINQ to XML类、LINQ to XML的一组嵌套元素和特性，把查询结果转换成XML。

```
foreach (var s in query)
```

```
{
```

```
    XElement storeElement = new XElement("store",  
        new XAttribute("name", s.Name),
```

```

        new XAttribute("address", s.Address),
        from stock in s.Stocks
        select new XElement("stock",
            new XAttribute("StockID", stock.StockId),
            new XAttribute("onHand",
                stock.OnHand),
            new XAttribute("onOrder",
                stock.OnOrder),
            new XElement("book",
                new XAttribute("title",
                    stock.Book.Title),
                new XAttribute("author",
                    stock.Book.Author)
            )// end book
        ) // end stock
    ); // end store
    WriteLine(storeElement);
}

```

这就使用LINQ和Entity Framework的全部功能，把19、20和21章的数据访问知识结合到一个程序中。

21.9 练习

(1) 修改第一个示例`BegVCSharp_21_1_CodeFirstDatabase`，提示用户输入书名和作者，把用户输入的数据存储到数据库中。

(2) 如果反复运行，第一个例子`BegVCSharp_21_1_CodeFirstDatabase`会创建重复的记录。修改例子，使其不创建重复的记录。

(3) 最后一个例子`BegVCSharp_21_3_XMLfromDatabase`生成的`BookContext`类所使用的关系名称不同于前面的示例`BegVCSharp_21_2_DatabaseRelations`。修改`BookContext`类，以使用相同的关系名。

(4) 使用`Code First`创建一个数据库，存储第19章的`GhostStories.xml`文件中的数据。

习题答案在附录A中。

21.10 本章要点

主题	要点
使用数据库	数据库是永久的、结构化的数据仓库。有许多不同类型的数据库，业务数据最常用的类型是关系数据库
Entity Framework	Entity Framework是一组.NET类，表示C#对象和关系数据库之间的对象-关系映射
如何使用Code First创建数据	在Entity Framework中使用Code First，可以使用对象-关系映射，直接从C#类和集合中创建数据库
如何给数据库使用LINQ	LINQ to Entities可使用与创建数据相同的Entity Framework类，支持在数据库上运行强大的查询功能
如何导航数据库关系	Entity Framework允许在数据库中通过使用C#代码中的虚拟属性和集合，创建和导航相关的实体
如何在数据库中创建和查询XML	可在单个查询中结合LINQ to Entities、LINQ to Objects和LINQ to XML，在数据库中构建XML

第 V 部分 其他技术

➤ 第22章 Windows Communication Foundation

➤ 第23章 通用应用程序

第22章 Windows Communication Foundation

本章内容：

- WCF的含义
- WCF概念
- WCF编程

本章源代码下载：

本章源代码的下载地址为
www.wrox.com/go/beginningvisualc#2015programming。从该网页的
Download Code选项卡中下载Chapter 22 Code后，可以找到与本章示例
对应的单独文件。

近年来，随着Internet的日益普及，Web服务得以迅速发展。Web服务就像一个Web站点，只不过是由计算机（而不是人）使用的。例如，除了浏览某个Web站点来查看自己喜欢的电视节目的信息，还可以使用一个桌面应用程序，通过某个Web服务提取相同的信息。这种方法的优点是，同一个Web服务可以被各种应用程序使用，其中也包括Web站点。而且，还可以编写自己的应用程序或Web站点来使用第三方的Web服务。例如，把自己喜欢的电视节目的信息与地图服务结合起来，以显

示该节目的拍摄现场。

.NET Framework支持Web服务已经有一段时间了。但在最近的版本中，它把Web服务与远程技术结合起来，创建出了Windows Communication Foundation（WCF），这是在应用程序之间进行通信的一种通用基础结构。

远程技术可在一个进程中创建对象实例，在另一个进程中使用它们，即使创建对象的计算机与使用对象的计算机不同。但这种技术仍有它自己的问题。远程技术是有限制的，而且刚入门的程序员要掌握它也不容易。

WCF从Web服务中提取了服务、独立于平台的SOAP消息传输等概念，把它们与远程技术中的宿主服务器应用程序和高级绑定功能结合在一起，所以可以将这种技术看成一个超集，包含了Web服务和远程技术，但比Web服务强大，比远程技术更易于掌握。使用WCF，可从简单的应用程序转向使用SOA（Service-Oriented Architecture，面向服务的体系结构）的应用程序。SOA意味着可以分散处理，并在需要时连接跨本地网络和Internet的服务和数据，使用分布式处理。

本章将学习如何在应用程序代码中创建和使用WCF服务。另外，本章也介绍了WCF的原理，这些知识同样重要，可以帮助理解其工作机制。

22.1 WCF的含义

WCF技术允许创建服务，可以跨进程、计算机和网络从其他应用程序访问这些服务。利用这些服务，可在多个应用程序中共享功能，提供数据源，或者抽象复杂进程。

WCF服务提供的功能也封装为该服务的方法，由该服务提供。每个方法——在WCF术语中称为“操作（operation）”——每个操作都有一个端点，用于交换数据。根据用于连接服务的网络和特定的要求，这种数据交换可能由一个或更多个协议定义。

在WCF中，端点可以有多个绑定，每个绑定都指定了一种通信方式。绑定还可以指定其他信息，例如，必须满足什么安全要求才能与端点通信。例如，绑定可能需要用户名和密码身份验证或者Windows用户账户令牌。在连接一个端点时，绑定使用的协议会影响所使用的地址，如后面所述。

一旦连接了一个端点，就可以使用SOAP消息与它通信。所使用的消息形式取决于所进行的操作和该操作收发消息所需的数据结构。WCF使用协定（contract）指定所有这些信息。通过与服务交换的元数据可以查找协定。用于找出服务信息的一种常用格式是Web Service Description Language（WSDL），它最初用于Web服务。不过，WCF服务还可以用其他方式描述。

注意： WCF的使用和设置方式变化多端。可能使用WCF来创建

REST（Representative State Transfer）服务。这些服务依赖简单HTTP请求在客户端和服务端之间通信，因此，与SOAP消息相比，它们更小。

识别出要使用的服务和端点，知道了要使用的绑定和需要依从的协定后，就可以与WCF服务通信，这与使用在本地定义的对象一样简单。与WCF服务通信可以是简单的单向事务、请求/响应消息，也可以是从通信信道任一端发出的全双工通信，还可以在需要时使用消息负载优化技术，如Message Transmission Optimization Mechanism（MTOM）来打包数据。

WCF服务在存储它的计算机上运行为许多不同进程中的一个。Web服务总是运行在IIS上，而WCF服务可以选择适合的宿主进程。可以使用IIS驻留WCF服务，也可以使用Windows服务或可执行程序。如果使用TCP在本地网络上与WCF服务通信，就不需要在运行服务的PC上安装IIS。

WCF框架允许定制本节介绍的几乎所有方面。但这是一个高级主题，本章仅使用.NET 4.5默认提供的技术。

了解WCF服务的基础知识后，下面将详细介绍这些概念。

22.2 WCF概念

本节描述WCF的如下方面：

- WCF通信协议
- 地址、端点和绑定
- 协定
- 消息模式
- 行为
- 驻留

22.2.1 WCF通信协议

如前所述，可以通过许多传输协议与WCF服务通信。在.NET 4.5 Framework中定义了5个协议：

- **HTTP：** 它允许与任何地方（包括跨Internet）的WCF服务通信。可以使用HTTP通信技术创建WCF Web服务。
- **TCP：** 如果正确配置了防火墙，它允许与本地网络或跨Internet的WCF服务通信。TCP比HTTP高效，功能也比较多，但配置起来更加复杂。
- **UDP：** 类似于TCP，也允许通过本地网络或Internet进行通信，但它的实现方式与TCP略有不同。这种实现允许服务同时向多个客户端广播消息。
- **命名管道：** 它允许与WCF服务通信，该WCF服务与调用代码位于

同一台计算机的不同进程上。

- **MSMQ:** 这是一种排队技术，允许应用程序发送的消息通过队列路由到目的地。MSMQ是一种可靠的消息传输技术，可以确保发送给队列的消息一定达到该队列。MSMQ还是一种异步技术，所以只有排在前面的消息都处理完了，服务仍有效时，才能处理当前消息。

这些协议常常允许建立安全连接。例如，可以使用HTTPS协议建立Internet上的安全SSL连接。TCP使用Windows安全架构为本地网络上的安全性能提供了更多的可能性。UDP则不支持安全性。

为了连接WCF服务，必须知道它在什么地方。这表示必须知道端点的地址。

22.2.2 地址、端点和绑定

用于服务的地址类型取决于所使用的协议。本章前面介绍的3个协议（不包括MSMQ）都需要格式化的服务地址：

- **HTTP:** HTTP协议的地址是URL，其格式很常见：`http://<server>:<port>/<service>`。对于SSL连接，也可以使用`https://<server>:<port>/<service>`。如果在IIS中驻留服务，<service>就是扩展名为.svc的文件。IIS地址可能包含比这个示例更多的子目录，即.svc文件之前有更多使用/字符分隔的部分。
- **TCP:** TCP的地址采用`net.tcp://<server>:<port>/<service>`形式。
- **UDP:** UDP的地址采用`soap.udp://<server>:<port>/<service>`。对于多播通信，需要为<server>使用特定的一些值，但是这超出了本

章的讨论范围。

- 命名管道：命名管道连接的地址与上述类似，但没有端口号。其形式是`net.pipe://<server>/<service>`。

服务的地址是一个基地址，它可用于为表示操作的端点创建地址。例如，在`net.tcp://<server>:<port>/<service>/operation1`上有一个操作。

例如，假定创建一个WCF服务，它有一个操作，绑定了前面介绍的3个协议，就可以使用下面的基地址：

```
http://www.mydomain.com/services/amazingservices/mygreatservice
net.tcp://myhugeserver:8080/mygreatservice
net.pipe://localhost/mygreatservice
```

接着就可以给操作使用下面的地址：

```
http://www.mydomain.com/services/amazingservices/mygreatservice
net.tcp://myhugeserver:8080/mygreatservice/greatop
net.pipe://localhost/mygreatservice/greatop
```

从.NET 4开始，可以给操作使用默认端点，而不必明确地配置它们。这简化了配置，如果需要使用标准端点地址（如上面的示例所示），这表现得尤其明显。

如前所述，绑定不仅指定了操作使用的传输协议，还可以指定在传输协议上通信的安全要求、端点的事务处理功能和消息编码等。

绑定提供了极大灵活性，所以.NET Framework提供了一些可用的预定义绑定。还可以把这些绑定用作起点，修改它们，得到需要的绑定类型。预定义绑定有一些必须遵循的原则。每种绑定类型都用

System.ServiceModel名称空间中的一个类表示。表22-1列出了最常用的绑定及其基本信息。

表22-1 绑定类型

绑定	说明
BasicHttpBinding	最简单的HTTP绑定，Web服务使用的默认绑定，它的安全功能有限，不支持事务处理
WSHttpBinding	HTTP绑定的一种较高级形式，可以使用WSE中引入的所有额外功能
WSDualHttpBinding	扩展了WSHttpBinding功能，包含双向通信功能。在双向通信中，服务器可以启动与客户端的通信，还可以进行一般的消息交换
WSFederationHttpBinding	扩展了WSHttpBinding功能，包含联合功能。联合功能允许第三方实现单点登录（single sign-on）和其他专用安全措施。这是一个高级主题，本章不予讨论
NetTcpBinding	用于TCP通信，允许配置安全性、事务处理等
NetNamedPipeBinding	用于命名管道的通信，允许配置安全性、事务处理等
NetMsmqBinding	这些绑定用于MSMQ，本章不予讨论
NetPeerTcpBinding	用于对等绑定，本章不予讨论
WebHttpBinding	用于使用HTTP请求（而不是SOAP消息）的Web服务
UdpBinding	允许绑定到UDP协议

这个表中的许多绑定类拥有可用于其他配置的类似属性。例如，它们有可用于配置超时值的属性。本章后面介绍编码时会详细讨论。

从.NET 4开始，端点的默认绑定因所用协议而异。这些默认绑定如表22-2所示。

表22-2 .NET的默认绑定

协议	默认绑定
HTTP	BasicHttpBinding
TCP	NetTcpBinding
UDP	UdpBinding
命名管道	NetNamedPipeBinding
MSMQ	NetMsmqBinding

22.2.3 协定

协定确定了WCF服务的用法。可以定义如下几种协定：

- 服务协定： 包含服务的一般信息和服务提供的操作的一般信息。例如，该协定可以包含服务使用的名称空间。在为SOAP消息定义模式时，服务使用唯一的名称空间，以免与其他服务冲突。
- 操作协定： 定义操作的用法，这包括操作方法的参数和返回类型，以及其他信息，例如，方法是否返回响应消息。

- 消息协定： 允许定制SOAP消息内部的信息格式化方式。例如，数据应包含在SOAP标头中还是SOAP消息体中。在创建必须与旧系统集成的WCF服务时，就可以使用消息协定。
- 错误协定： 定义操作可能返回的错误。使用.NET客户端程序时，错误会导致可以捕获的异常，并以通常方式处理。
- 数据协定： 如果使用复杂类型，如用户定义的结构和对象（作为操作的参数或返回类型），就必须为这些类型定义数据协定。数据协定根据通过属性显示的数据来定义类型。

一般使用特性把协定添加到服务类和方法中，如本章后面所述。

22.2.4 消息模式

上一节提到，操作协定可以定义操作是否返回一个值，`WSDualHttpBinding`允许进行双向通信。这些都是消息模式。消息模式有3种类型：

- 请求/响应消息传输： 交换消息的“一般”方式，每个发送给服务的消息都会产生一个发送给客户端的响应。这并不意味着客户端必须要等待响应，因为可以用一般方式异步调用操作。
- 单向消息传输： 消息从客户端传输给WCF操作，但服务器不发送响应。
- 双向消息传输： 一种较高级的模式，客户端可以用作服务器，服务器也可以用作客户端。启动后，双向消息传输允许客户端和服务端彼此发送消息，这些消息可能没有响应。

本章后面将使用这些消息模式。

22.2.5 行为

行为（behavior）是把没有直接提供给客户端的其他配置应用于服务和操作的方式。给服务添加行为，可以控制宿主进程如何实例化和使用行为，行为如何参与事务处理，在服务中如何解决多线程问题等。操作行为可以控制在操作执行过程中是否使用模仿功能，各个操作行为如何影响事务处理等。

从.NET 4开始，可在不同的级别上指定默认行为，而不必给每个服务和操作指定每个行为的各个方面。还可在需要时提供默认设置和重写设置，减少所需的配置量。

22.2.6 驻留

本章的引言曾提到，WCF服务可以存储在几个不同进程中，包括：

- **Web服务器：** 驻留在IIS的WCF服务是WCF提供的最接近Web服务的服务。还可以使用WCF服务中的高级功能和安全特性，这些功能和特性很难在Web服务中实现，也可以集成IIS特性，如IIS安全特性。
- **可执行文件：** 可以把WCF服务驻留在.NET中创建的任意应用程序类型中，如控制台应用程序、Windows窗体应用程序和WPF应用程序。
- **Windows服务：** 可以把WCF服务驻留在Windows服务中，这意味着可以使用Windows服务提供的有用特性，包括自动启动和错误恢复。

- **Windows Activation Service (WAS)**：专门用于驻留WCF服务，基本上是IIS的一个简化版本，可以在任何没有IIS的地方使用。

上述列表中的两个选项IIS和WAS为WCF服务提供了有用的特性，例如激活、进程回收和对象池。如果使用另外两个驻留选项，WCF服务就是自驻留的。我们偶尔会自驻留服务，以进行测试，但最好创建自驻留、产品级的服务。例如，假定不允许在运行服务的电脑上安装Web服务器。如果服务运行在域控制器上，或者公司的本地策略只是禁止运行IIS，就可以把服务驻留在Windows服务上，它会工作得很好。

22.3 WCF编程

前面介绍了基础知识，下面开始编写一些代码。本节首先看一个在Web服务器上驻留的简单WCF服务和一个控制台客户端程序。介绍了所创建的代码结构后，学习WCF服务和客户端应用程序的基本结构。此后详细探讨一些重要主题：

- 定义WCF服务协定
- 自驻留的WCF服务

试一试：一个简单的**WCF**服务和客户端程序：

Ch22Ex01Client

- （1）在C:\BegVCSharp\Chapter22目录中创建一个新的WCF服务应用程序项目Ch22Ex01。
- （2）在解决方案中添加一个控制台应用程序Ch22Ex01Client。
- （3）在Build菜单上单击Build Solution选项。
- （4）在Solution Explorer中右击Ch22Ex01Client项目，选择Add Service Reference选项。
- （5）在Add Service Reference对话框中，单击Discover。

(6) 启动开发Web服务器，加载WCF服务的信息后，展开该引用，查看其细节，注意服务中有两个方法：GetData和GetDataUsingDataContract。

(7) 单击OK按钮，添加服务引用。

(8) 在Ch22Ex01Client应用程序中修改Program.cs中的代码，如下所示：

```
using Ch22Ex01Client.ServiceReference1;
using static System.Console;
namespace Ch22Ex01Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Title = "Ch22Ex01Client";
            string numericInput = null;
            int intParam;
            do
            {
                WriteLine("Enter an integer and press enter to call th
                numericInput = ReadLine();
            }
            while (!int.TryParse(numericInput, out intParam));
            Service1Client client = new Service1Client();
            WriteLine(client.GetData(intParam));
```

```
        WriteLine("Press an key to exit.");  
        ReadKey();  
    }  
}  
}
```

(9) 在Solution Explorer中右击Ch22Ex01Client项目，选择Set as StartUp Project选项。

(10) 运行应用程序。在控制台应用程序窗口中输入一个数字，按下回车键，结果如图22-1所示。

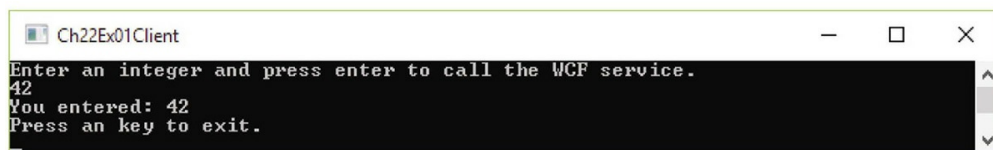


图22-1

(11) 退出应用程序，在Solution Explorer中右击Ch22Ex01项目中的Service1.svc文件，单击View in Browser。

(12) 查看窗口中的信息。

(13) 单击Web页面顶部的链接，查看服务的WSDL。现在还不需要了解WSDL文件中的所有内容的含义。

示例的说明

这个示例中创建了一个驻留在Web服务器上的简单Web服务和控制

台客户端程序。我们为WCF服务项目使用了默认的VS模板，这意味着不必自己添加任何代码，而是可以使用这个默认模板中定义的一个操作GetData()。对于这个示例，使用什么操作并不重要，而应关注代码的结构及其工作方式。

首先看看服务器项目Ch22Ex01，它包含：

- Service1.svc文件，它定义了服务的宿主。
- 类定义CompositeType，它定义了服务使用的数据协定（位于IService1.cs代码文件中）。
- 接口定义IService1，它定义了服务协定和两个操作协定。
- 类定义Service1，它实现IService1接口，定义了服务的功能（位于Service1.svc.cs代码文件中）。
- 配置段<system.serviceModel>（在Web.config中），它配置了服务。

Service1.svc文件包含如下代码行。要查看这行代码，应在Solution Explorer中右击该文件，再单击View Markup：

```
<%@ ServiceHost Language="C#" Debug="true" Service="Ch22Ex01.S  
CodeBehind="Service1.svc.cs" %>
```

这是一个ServiceHost指令，用于告诉Web服务器（本例是Web开发服务器，但该指令也可应用于IIS）把什么服务存储在这个地址上。定义服务的类在Service特性中声明，定义这个类的代码文件在CodeBehind特性中声明。这个指令是必需的，以获得Web服务器的驻留功能，如前面几节所述。

显然，没有驻留在Web服务器上的WCF服务不需要这个文件。本章

后面将学习自驻留的WCF服务。

接着在IService1.cs文件中定义数据协定CompositeType。从代码中可以看出，数据协定只是一个类定义，在类定义中包含了DataContract特性，在类成员上包含了DataMember特性：

```
[DataContract]
public class CompositeType
{
    bool boolValue = true;
    string stringValue = "Hello ";
    [DataMember]
    public bool BoolValue
    {
        get { return boolValue; }
        set { boolValue = value; }
    }
    [DataMember]
    public string StringValue
    {
        get { return stringValue; }
        set { stringValue = value; }
    }
}
```

这个数据协定通过元数据提供给客户端应用程序（查看示例中的WSDL文件，就会看到这些元数据）。这允许客户端应用程序定义一个类型，该类型可以序列化到窗体上，该窗体又可以由服务反序列化到

CompositeType对象上。客户端程序不需要知道这个类型的实际定义，实际上，客户端程序使用的类可以有不同的实现代码。定义数据协定的这种方式虽简单但非常强大，允许在WCF服务及其客户端程序之间交换复杂的数据结构。

IService1.cs文件还包含服务协定，该服务协定定义为带有ServiceContract特性的接口。这个接口也在服务元数据中进行了完整描述，并可在客户端应用程序中重建。接口成员构成了服务的操作，每个操作都应用OperationContract特性创建一个操作协定。示例代码包含两个操作，其中一个操作使用了前面的数据协定：

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    string GetData(int value);
    [OperationContract]
    CompositeType GetDataUsingDataContract(CompositeType compos
}
```

前面介绍的4个协定定义特性都可以用特性进一步配置，如下一节所述。实现服务的代码与其他类定义类似：

```
public class Service1 : IService1
{
    public string GetData(int value)
    {
        return string.Format("You entered: {0}", value);
    }
}
```

```

    }
    public CompositeType GetDataUsingDataContract(CompositeType
    {
    ...
    }
}

```

注意这个类定义不需要继承自特定类型，也不需要任何特定的特性，只需实现定义了服务协定的接口。实际上，可以在这个类及其成员中添加特性，以指定行为，但这些都不是强制的。

把服务的实现代码（类）和服务协定（接口）分开的效果极佳。客户端程序不需要了解类的任何信息，类包含的功能可能远远超过了服务实现的功能。一个类甚至可以实现多个服务协定。

最后来分析Web.config文件中的配置。在配置文件中，WCF服务的配置是从.NET远程技术中提取出来的一个特性，可以处理所有类型的WCF服务（非自驻留的服务和自驻留的服务）和WCF服务的客户端程序（稍后介绍）。这个配置的词汇允许把任何配置应用于服务，甚至可以扩展其语法。

WCF配置代码包含在Web.config或app.config文件的配置段<system.serviceModel>中。这个示例使用了默认值，所以没有进行很多服务配置。在Web.config文件中，配置段包含一个子段，它为服务行为<behaviors>重写了默认值。Web.config中<system.serviceModel>配置段的代码如下（为简洁起见，删除了注释）：

```
<system.serviceModel>
```

```
<behaviors>
  <serviceBehaviors>
    <behavior>
      <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
      <serviceDebug includeExceptionDetailInFaults="false" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
```

这个配置段可在<behavior>子段中定义一个或多个行为，这些行为可以在多个其他元素上重用。可以给<behavior>段指定一个名称，以便进行重用（这样就可以在其他地方引用它），也可以不指定名称来使用（如本例所示），以指定重写默认的行为设置。

注意： 如果使用了非默认的配置，在<system.serviceModel>中就会包含一个<services>段，其中包含一个或多个<service>子段，<service>段又可以包含<endpoint>子段，每个<endpoint>子段都定义了服务的一个端点。实际上，所定义的端点是服务的基端点。可从中推断出操作的端点。

在Web.config中，重写的一个默认行为如下：

```
<serviceDebug includeExceptionDetailInFaults="false"/>
```

这个设置可以是true，在传输给客户端程序的任意错误中提供异常

详情，通常只允许在开发过程中传输这些异常信息。

在Web.config中，另一个默认的重写行为与元数据相关。元数据允许客户端程序获得WCF服务的描述。默认配置为服务定义了两个默认端点，一个端点由客户端程序用于访问服务；另一个端点用于获得服务的元数据。在Web.config文件中，可以禁用这个功能，如下所示：

```
<serviceMetadata httpGetEnabled="false"

httpsGetEnabled="false"

/>
```

另外，还可以完全删除这行配置代码，因为默认行为不允许交换元数据。

如果在本例中尝试禁用这个功能，并不能阻止客户端程序访问服务，因为客户端程序已经在添加服务引用时获得了需要的元数据。但禁用元数据会禁止其他客户端程序使用Add Service Reference工具访问这个服务。一般情况下，生产环境中的Web服务不需要提供元数据，所以应在开发阶段完成后禁用这个功能。

如果没有元数据，访问Web服务的另一种常见方式是在一个独立的程序集中为WCF服务定义协定，由宿主项目和客户项目引用。接着客户端程序就可以直接使用这些协定生成一个代理，而不是通过元数据来访问服务。

前面介绍了WCF服务的代码，现在看看客户端程序，尤其是使用Add Service Reference工具做了什么。注意在Solution Explorer中，客户端程序包含一个文件夹Service References，如果展开该文件夹，就会看到一项ServiceReference1，它是添加引用时选用的名称。

Add Service Reference工具创建了访问服务需要的所有类。这包括服务的代理类，服务的代理类包含服务的所有操作方法（Service1Client），以及从数据协定中生成的客户端类（CompositeType）。

注意： 也可以浏览Add Service Reference工具生成的代码（显示项目中的所有文件，包括隐藏的文件），但目前最好不要浏览代码，因为有许多容易引起混淆的代码。

该工具还为项目添加了一个配置文件app.config，这个配置定义了两个内容：

- 服务端点的绑定信息
- 端点的地址和协定

从服务描述中提取绑定信息：

```
<configuration>  
  <system.serviceModel>
```

```
<bindings>
```

```
<basicHttpBinding>
```

```
<binding name="BasicHttpBinding_IService1" />
```

```
</basicHttpBinding>
```

```
</bindings>
```

这个绑定、服务的基地址（这是Web服务器存储的服务的.svc文件地址）和协定的客户端版本IService1在端点配置中使用：

```
<client>
```

<endpoint address="http://localhost:49227/Service1.svc"

binding="basicHttpBinding"

bindingConfiguration="BasicHttpBinding_IService1"

contract="ServiceReference1.IService1"

name="BasicHttpBinding_IService1" />

</client>

</system.serviceModel>

</configuration>

这段代码删除了整个<bindings>段和<endpoint>元素的bindingConfiguration特性，这表示客户端程序将使用默认的绑定配置。

<binding>元素的名称是BasicHttpBinding_IService1，这里包含它是为了定制绑定的配置。这里可用的配置有很多，包括超时设置、消息大小限制和安全设置等。如果服务项目把这些配置指定为非默认值，就可以在app.config文件中看到它们，因为它们会被复制到这个文件中。只有绑定配置匹配时，客户端程序才能与服务通信。本章不深入探讨WCF服务配置。

这个示例介绍了许多基础知识，下面总结一下前面的内容：

- WCF定义
 - 服务由服务协定接口定义，其中包括操作协定成员
 - 服务在实现了服务协定接口的类中实现
 - 数据协定只是使用数据协定特性的类型定义
- WCF服务配置
 - 可使用配置文件（Web.config或app.config）来配置WCF服务
- WCF Web服务器驻留：
 - Web服务器驻留把.svc文件用作服务基地址
- WCF客户程序配置：
 - 可使用配置文件（web.config或app.config）来配置WCF服务的客户程序

下一节将详细介绍协定。

22.3.1 WCF测试客户端程序

上面的示例创建了服务和客户端程序，说明了基本WCF体系结构的工作原理，以及如何归档WCF服务的配置。但实际要使用的客户端应用程序会比较复杂，也难以正确测试服务。

为便于开发WCF服务，VS提供了一个测试工具，可用于确保WCF操作正常工作。这个工具会自动配置为处理WCF服务项目，所以如果运行项目，该工具就会显示出来。只需要确保要测试的服务（即.svc文件）设置为WCF服务项目的启动页面即可。另外，也可以把测试客户端程序作为独立的应用程序运行。在64位操作系统上，测试客户端程序位于C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\WcfTestClient.exe。

如果使用32位操作系统，该路径是相同的，只是根文件夹是Program Files。

可使用该工具调用服务操作，还可以用其他方式检查服务。如下面的示例所示。

试一试：使用WCF测试客户端程序：Ch22Ex01\Web.config

(1) 打开上一个示例中的WCF Service Application项目Ch22Ex01。

(2) 在Solution Explorer中右击Service1.svc服务，然后单击Set As Start Page。

(3) 在Solution Explorer中右击Ch22Ex01项目，然后单击Set As StartUp Project。

(4) 在Web.config中，确保启用元数据。

```
<serviceMetadata httpGetEnabled="true"
```

```
httpsGetEnabled="true"
```

```
/>
```

(5) 运行应用程序。WCF测试客户端程序就会显示出来。

(6) 在测试客户端程序的左面板上双击Config File。用于访问服务的配置文件就会显示在右面板上。

(7) 在左面板上双击GetDataUsingDataContract()操作。

(8) 在右面板上把BoolValue的值改为True，StringValue改为Test String，再单击Invoke。

(9) 如果显示了安全提示对话框，单击OK按钮确认把信息发送给服务。

(10) 显示操作的结果，如图22-2所示。

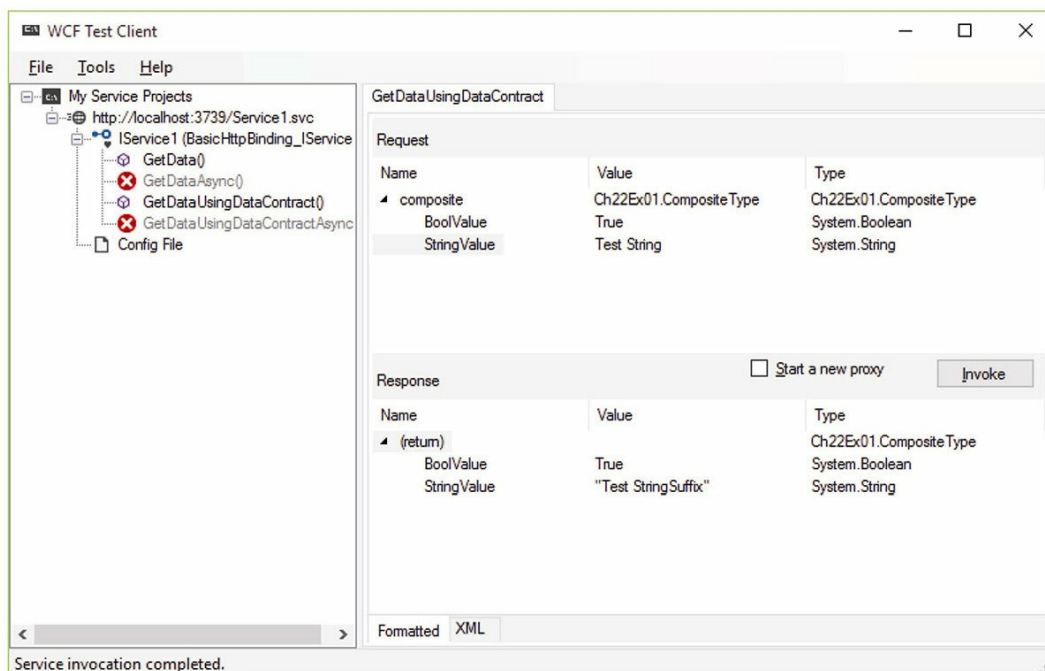


图22-2

(11) 单击XML标签页，查看请求和响应的XML。

(12) 关闭WCF测试客户端程序，这会停止VS中的调试。

示例的说明

这个示例使用WCF测试客户端程序在上一个示例创建的服务上检查和调用操作。首先注意，服务的加载有一点儿延迟，这是因为测试客户端程序必须检查服务，以确定其功能。这个检查过程使用与Add Service Reference工具相同的元数据，所以必须确保元数据是可用的（在上一个示例中可能禁用了元数据）。检查完毕后，在工具的左面板上就会显示服务及其操作。

接着查看用于访问服务的配置。与上一个示例中的客户端应用程序

一样，这些配置也是从服务的元数据中自动生成的，且包含在与服务相同的代码中。如有必要，可以通过该工具编辑这个配置文件，方法是右击Config File项，单击Edit WCF Configuration。图22-3是该配置的一个示例，其中包含了本章前面提到的绑定配置选项。

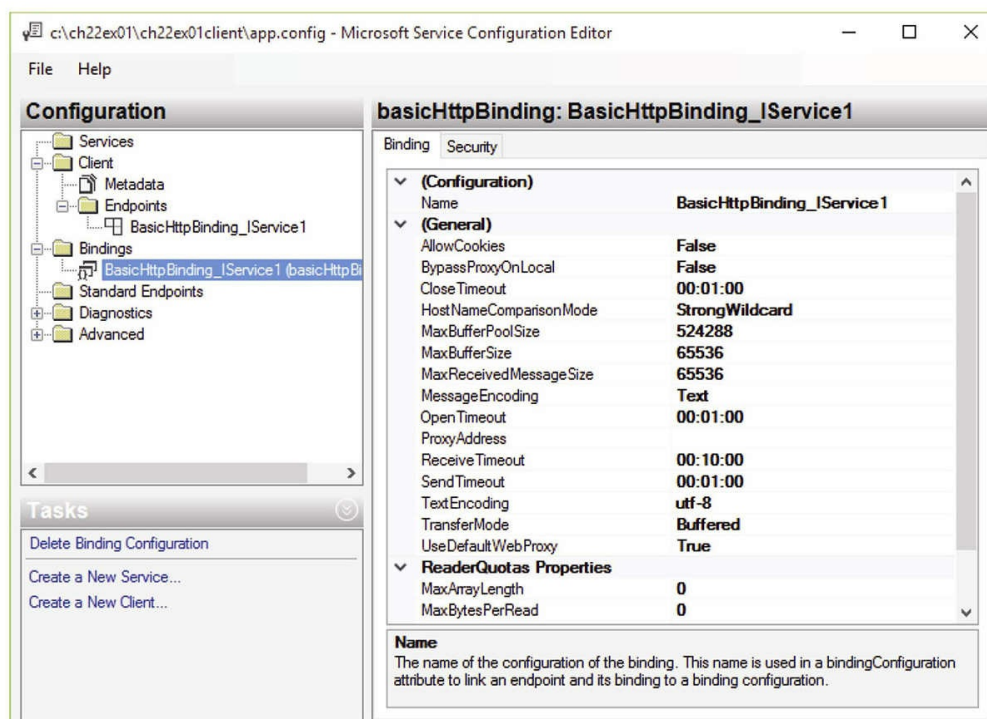


图22-3

最后调用了一个操作。测试客户端程序允许输入要使用的参数，并调用方法，然后显示结果，所有这些都不需要编写任何客户代码。我们还查看了为获得结果而发送和接收的XML，这些信息的技术性很强，但在调试比较复杂的服务时，这些信息是绝对必需的。

22.3.2 定义WCF服务协定

从前面的示例可以了解到，通过WCF体系结构，可以结合使用类、接口和特性来方便地为WCF服务定义协定。本节将深入介绍这种技术。

1. 数据协定

要给服务定义数据协定，需要把DataContractAttribute特性应用于类定义。这个特性在名称空间System.Runtime.Serialization名称空间中。可使用表22-3所示的属性配置它。

表22-3 DataContractAttribute的属性

属性	说明
Name	用不同于类定义的名称来命名数据协定。这个名称在SOAP消息和服务元数据定义的客户端数据对象上使用
Namespace	指定数据协定在SOAP消息中使用的名称空间
IsReference	影响序列化对象的方式。如果设置为true，那么即使多次引用某个对象实例，仍然只序列化该对象实例一次，有些情况下，这可能非常重要。默认值是false

当需要与已有的SOAP消息格式交互操作时，Name和Namespace属性非常重要（其他协定的类似名称的属性也是同理），但在其他情况下很可能不需要使用它们。

数据协定中的每个类成员都必须使用DataMemberAttribute特性，它在名称空间System. Runtime.Serialization中。这个特性具有表22-4所示的属性。

表22-4 DataMemberAttribute的属性

属性	说明
Name	指定序列化时数据成员的名称（默认为成员名称）
IsRequired	指定成员是否必须显示在SOAP消息中
Order	int值，指定序列化或反序列化成员的顺序，如果一个成员必须在另一个成员之前出现，这个顺序就是必需的。先处理Order较低的成员
EmitDefaultValue	将其设置为false时，如果成员的值是默认值，就禁止该成员包含在SOAP消息中

2. 服务协定

把System.ServiceModel.ServiceContractAttribute特性应用于接口定义，就定义了服务协定。表22-5所示的属性可用于定制服务协定。

表22-5 ServiceContractAttribute的属性

属性	说明
Name	按照WSDL中<portType>元素中的定义，指定服务协定的名称
Namespace	定义WSDL中<portType>元素使用的服务协定的名称空间
ConfigurationName	在配置文件中使用的服务协定名称

HasProtectionLevel	指定服务使用的消息是否有明确定义的保护级别。保护级别允许签名消息，或者签名和加密消息
ProtectionLevel	保护级别，用于保护消息
SessionMode	确定是否为消息启用会话。如果使用会话，就可以确保关联上发送给服务的不同端点的消息，即它们使用同一个服务实例，因此可以共享状态
CallbackContract	对于双向消息传输，客户端提供了协定和服务。这是因为，如前所述，双向通信中的客户端也用作服务器。这个属性允许指定客户端使用的协定

3. 操作协定

在定义服务协定的接口中，应用 `System.ServiceModel.OperationContractAttribute` 特性，就可以把成员定义为操作。这个特性具有表22-6所示的属性。

表22-6 `OperationContractAttribute` 的属性

属性	说明
Name	指定服务操作的名称。默认为成员名称
IsOneWay	指定操作是否返回一个响应。如果把它设置为 <code>true</code> ，则客户端不等待操作完成，就会继续执行
AsyncPattern	如果设置为 <code>true</code> ，操作就会实现为两个方法： <code>Begin<methodName>()</code> 和 <code>End<methodName>()</code> ，

	这两个方法可用于异步调用操作
HasProtectionLevel	参见表22-5
ProtectionLevel	参见表22-5
IsInitiating	如果使用会话，这个属性就确定调用这个操作是否可以启动新会话
IsTerminating	如果使用会话，这个属性就确定调用这个操作是否会中断当前会话
Action	如果使用寻址功能（WCF服务的一个高级功能），操作就有一个关联的动作名称，通过这个属性可以指定该名称
ReplyAction	同上，但为操作的响应指定动作名称

注意： 在.NET 4.5 Framework中，添加一个服务引用时，无论AsyncPattern是否设置为true，VS都会生成用于调用该服务的异步代理方法。这些方法带有后缀Async，它们使用了.NET 4.5中新引入的异步技术，并且只是从调用代码的角度看才是异步的。在内部，它们调用的是同步的WCF操作。

4. 消息协定

前面的示例中没有使用消息协定规范。如果使用消息协定，就应定义一个表示消息的类，再给类应用MessageContractAttribute特性。接着给这个类的成员应用MessageBodyMemberAttribute、

MessageHeaderAttribute或MessageHeaderArrayAttribute特性。所有这些特性都在System.ServiceModel名称空间中。除非要高度控制WCF服务使用的SOAP消息，否则一般不会使用消息协定，所以这里不详细讨论它。

5. 误协定

如果客户端应用程序可以使用特定的异常类型，如定制异常，就可以给可能生成该异常的操作应用System.ServiceModel.FaultContractAttribute特性。

试一试：WCF协定：Ch22Ex02Contracts

(1) 在C:\BegVCSharp\Chapter22目录中创建一个新WCF服务应用程序项目Ch22Ex02。

(2) 给解决方案添加一个类库项目Ch22Ex02Contracts，删除Class1.cs文件。

(3) 在Ch22Ex02Contracts项目中添加对System.Runtime.Serialization.dll和System.ServiceModel.dll程序集的引用。

(4) 在Ch22Ex02Contracts项目中添加Person类，修改Person.cs中的代码，如下所示：

```

using System.Runtime.Serialization;
namespace Ch22Ex02Contracts
{
    [DataContract]
    public class Person
    {
        [DataMember]
        public string Name { get; set; }
        [DataMember]
        public int Mark { get; set; }
    }
}

```

(5) 在Ch22Ex02Contracts项目中添加IAwardService接口，修改IAwardService.cs中的代码，如下所示：

```

using System.ServiceModel;
namespace Ch22Ex02Contracts
{
    [ServiceContract(SessionMode = SessionMode.Required)]
    public interface IAwardService
    {
        [OperationContract(IsOneWay = true, IsInitiating = true)]
        void SetPassMark(int passMark);
        [OperationContract]
        Person[] GetAwardedPeople(Person[] peopleToTest);
    }
}

```

}

(6) 对于Ch22Ex02项目，添加对Ch22Ex02Contracts项目的引用。

(7) 删除Ch22Ex02项目中的IService1.cs和服务1.svc。

(8) 在Ch22Ex02中添加一个新的WCF服务AwardService。

(9) 删除Ch22Ex02项目中的IAwardService.cs文件。

(10) 修改AwardService.svc.cs文件中的代码，如下所示：

```
using System.Collections.Generic;
using Ch22Ex02Contracts;
namespace Ch22Ex02
{
    public class AwardService : IAwardService
    {
        private int passMark;
        public void SetPassMark(int passMark)
        {
            this.passMark = passMark;
        }
        public Person[] GetAwardedPeople(Person[] peopleToTest)
        {
            List<Person> result = new List<Person>();
            foreach (Person person in peopleToTest)
            {
                if (person.Mark > passMark)
```

```

    {
        result.Add(person);
    }
}
return result.ToArray();
}
}
}

```

(11) 修改Web.config中的服务配置段，如下所示：

```

<system.serviceModel>
    <protocolMapping>

```

```

        <add scheme="http" binding="wsHttpBinding" />

```

```

    </protocolMapping>

```

```

    ...

```

```

</system.serviceModel>

```

(12) 打开Ch22Ex02的项目属性。在Web部分，记住宿主设置中使

用的端口号。如果尚未安装IIS，则可以在Visual Studio Development Server中设置一个特定的端口。

(13) 在解决方案中添加一个新的控制台项目Ch22Ex02Client，把它设置为启动项目。

(14) 在Ch22Ex02Client项目中添加对System.ServiceModel.dll程序集和Ch22Ex02Contracts项目的引用。

(15) 在Ch22Ex02Client项目中修改Program.cs中的代码，如下所示（确保使用了前面在EndpointAddress构造函数中获得的端口号，示例代码使用了49284）：

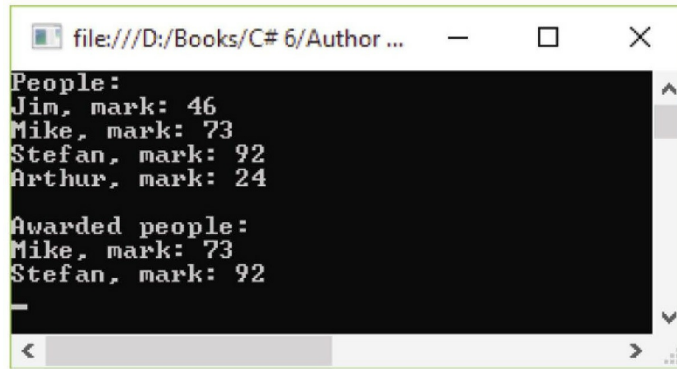
```
using System;
using static System.Console;
using System.ServiceModel;
using Ch22Ex02Contracts;
namespace Ch22Ex02Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Person[] people = new Person[]
            {
                new Person { Mark = 46, Name="Jim" },
                new Person { Mark = 73, Name="Mike" },
                new Person { Mark = 92, Name="Stefan" },
            }
        }
    }
}
```

```

        new Person { Mark = 24, Name="Arthur" }
    };
    WriteLine("People:");
    OutputPeople(people);
    IAwardService client = ChannelFactory<IAwardService>.Create
        new WSHttpBinding(),
        new EndpointAddress("http://localhost:38831/AwardService");
    client.SetPassMark(70);
    Person[] awardedPeople = client.GetAwardedPeople(people);
    WriteLine();
    WriteLine("Awarded people:");
    OutputPeople(awardedPeople);
    ReadKey();
}
static void OutputPeople(Person[] people)
{
    foreach (Person person in people)
        WriteLine("{0}, mark: {1}", person.Name, person.Mark);
}
}
}

```

(16) 如果使用了IIS，直接运行应用程序即可。如果使用了开发服务器，必须确保该服务的开发服务器正在运行，所以需要首先运行服务项目。为此，可以Ch22Ex02项目设置为启动项目，然后按Ctrl+F5键。这将启动该服务，但不进行调试。然后把Ch22Ex02Client项目设置为启动项目，再按下F5键。结果如图22-4所示。



```
file:///D:/Books/C# 6/Author ...
People:
Jin, mark: 46
Mike, mark: 73
Stefan, mark: 92
Arthur, mark: 24

Awarded people:
Mike, mark: 73
Stefan, mark: 92

-
```

图22-4

示例的说明

这个示例在类库项目中创建了一系列协定，在WCF服务和客户端程序中使用了这个类库。与前面的示例一样，这个服务也驻留在Web服务器上。这个服务的配置也被减少到最低程度。

在这个示例中，主要区别是客户端程序不需要元数据，因为客户端程序可以访问协定程序集。客户端程序不是从元数据中生成一个代理类，而是通过另一种方法获得服务协定接口的引用。这个示例中另一个值得注意的地方是使用会话维护服务中的状态，这需要WSHttpBinding绑定，而不是BasicHttpBinding绑定。

这个示例使用的数据协定是一个简单的类Person，它有一个string属性Name和一个int属性Mark。使用的DataContractAttribute特性和DataMemberAttribute特性没有进行定制，也不需要给这个协定重复迭代代码。

定义服务协定时，给IAwardService接口应用了ServiceContractAttribute特性。这个特性的SessionMode属性设置为

SessionMode.Required, 因为这个服务需要状态:

```
[ServiceContract(SessionMode=SessionMode.Required)]  
public interface IAwardService  
{
```

第一个操作协定SetPassMark()设置状态, 因此其OperationContractAttribute的IsInitiating属性设置为true。这个操作不返回任何值, 所以将IsOneWay设置为true, 把操作定义为单向操作:

```
[OperationContract(IsOneWay=true,IsInitiating=true)]  
void SetPassMark(int passMark);
```

另一个操作协定GetAwardedPeople()不需要进行任何定制, 使用前面定义的数据协定:

```
[OperationContract]  
Person[] GetAwardedPeople(Person[] peopleToTest);  
}
```

这两个类型Person和IAwardService都可以用于服务和客户端程序。服务在AwardService类型中实现了IAwardService协定, 它不包含任何特殊代码。这个类与前面的服务类的唯一区别是, 这个类是有状态的。这是允许的, 因为定义了一个会话, 来关联来自客户端程序的消息。

为确保服务使用WSHttpBinding绑定, 给服务添加了如下Web.config:

```
<protocolMapping>  
  <add scheme="http" binding="wsHttpBinding" />
```

```
</protocolMapping>
```

这重写了HTTP绑定的默认映射。另外，也可以手工配置服务，保留已有的默认配置，但这个重写的配置要简单得多。注意此类重写配置会应用于项目中的所有服务。如果项目中有多个服务，就必须确保每个服务都能接受这个绑定。

客户端程序比较有趣，主要是因为下面这行代码：

```
IAwardService client = ChannelFactory<IAwardService>.CreateChannel(
    new WSHttpBinding(),
    new EndpointAddress("http://localhost:38831/AwardService"));
```

客户端程序没有用app.config文件来配置与服务的通信，也没有从元数据中定义代理类，来与服务通信。而是通过ChannelFactory<T>.CreateChannel()方法创建代理类。这个方法创建了一个实现IAwardService客户端程序的代理类，但在后台生成的类与服务通信，就像前面通过元数据生成的代理一样。

注意： 如果通过ChannelFactory<T>.CreateChannel()方法创建代理类，通信信道就默认为在1分钟后超时，导致通信错误。使连接一直处于激活状态有许多方式，但这些都超出了本章的讨论范围。

采用这种方式创建代理类是一种非常有用的技术，可以快速生成客户端应用程序。

22.3.3 自驻留的WCF服务

本章前面介绍了驻留在Web服务器上的WCF服务。它们可以在Internet上通信，但对于本地网络通信而言，这并不是最高效的方式。一方面，需要用计算机上的Web服务器驻留服务；另一方面，在应用程序的体系结构上出现一个独立的WCF服务可能并不合适。

因此应使用自驻留的WCF服务。自驻留的WCF服务存在于创建它的进程中，而不存在于特别建立的主机应用程序（如Web服务器）的进程中。这意味着可以使用控制台应用程序或Windows应用程序驻留服务了。

要建立自驻留的WCF服务，需要使用System.ServiceModel.ServiceHost类。用要驻留的服务类型或服务类的一个实例来实例化这个类。通过属性或方法可以配置服务宿主，也可以通过配置文件来配置。实际上，宿主进程（如Web服务器）使用ServiceHost实例执行该驻留任务。自驻留时，区别是直接与此类交互操作。但在宿主应用程序的app.config文件中，<system.serviceModel>段中的配置使用的语法与本章前面的配置段中的相同。

可以通过任意协议提供自驻留的WCF服务，但是一般在这种类型的应用程序中使用TCP或命名管道绑定。通过HTTP访问的服务常常位于Web服务器进程中，因为可以获得Web服务器提供的额外功能，如安全性等。

如果要驻留MyService服务，可使用下面的代码创建ServiceHost的一个实例：

```
ServiceHost host = new ServiceHost(typeof(MyService));
```

如果要存储MyService的实例MyServiceObject，可以编写如下代码，创建ServiceHost的一个实例：

```
MyService myServiceObject = new MyService();  
ServiceHost host = new ServiceHost(myServiceObject);
```

警告： 只有配置了服务，使调用总是可以路由到同一个对象实例上，才能在ServiceHost中驻留服务实例。为此，必须给服务类应用ServiceBehaviorAttribute特性，将这个特性的InstanceContextMode属性设置为InstanceContextMode.Single。

创建ServiceHost实例后，就可以通过属性配置服务及其端点和绑定。另外，如果把配置放在.config文件中，就会自动配置ServiceHost实例。

有了配置好的ServiceHost实例后，为了开始驻留服务，应使用ServiceHost.Open()方法。同样，通过ServiceHost.Close()方法可以停止驻留服务。第一次驻留TCP绑定的服务时，如果启用它，可能会收到Windows防火墙服务发出的一个警告，因为它阻塞了默认的TCP端口。只有给这个服务打开TCP端口，才能开始监听该端口。

下面的示例使用自驻留技术通过WCF服务提供WPF应用程序的一些功能。



试一试：自驻留的WCF服务：Ch22Ex03

(1) 在C:\BegVCSharp\Chapter22目录中创建一个新的WPF应用程序Ch22Ex03。

(2) 使用Add New Item向导给项目添加一个新的WCF服务AppControlService。

(3) 修改MainWindow.xaml中的代码，如下所示：

```
<Window x:Class="Ch22Ex03.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Stellar Evolution" Height="450" Width="430"
        Loaded="Window_Loaded" Closing="Window_Closing">

    <Grid Height="400" Width="400" HorizontalAlignment="Center"
        VerticalAlignment="Center">
```



```
<Rectangle Fill="Black" RadiusX="20" RadiusY="20"
```

```
StrokeThickness="10">
```

```
<Rectangle.Stroke>
```

```
<LinearGradientBrush EndPoint="0.358,0.02"
```

```
StartPoint="0.642,0.98">
```

```
<GradientStop Color="#FF121A5D" Offset="0" />
```

```
<GradientStop Color="#FFB1B9FF" Offset="1" />
```

</LinearGradientBrush>

</Rectangle.Stroke>

</Rectangle>

<Ellipse Name="AnimatableEllipse" Stroke="{x:Null}" Height="100" Fill="{x:Null}"

Width="0" HorizontalAlignment="Center"

VerticalAlignment="Center">

<Ellipse.Fill>

<RadialGradientBrush>

<GradientStop Color="#FFFFFFFF" Offset="0" />

<GradientStop Color="#FFFFFFFF" Offset="1" />

</RadialGradientBrush>

</Ellipse.Fill>

<Ellipse.Effect>

```
<DropShadowEffect ShadowDepth="0" Color="#FFFFFFFF"
```

```
BlurRadius="50" />
```

```
</Ellipse.Effect>
```

```
</Ellipse>
```

```
</Grid>
```

```
</Window>
```

（4）修改MainWindow.xaml.cs中的代码，如下所示：

```
using System.Windows.Shapes;  
using System.ServiceModel;
```

```
using System.Windows.Media.Animation;
```

```
namespace Ch22Ex03
```

```
{
```

```
    ///<summary>
```

```
    /// Interaction logic for MainWindow.xaml
```

```
    ///</summary>
```

```
    public partial class MainWindow : Window
```

```
    {
```

```
        private AppControlService service;
```

```
private ServiceHost host;
```

```
    public MainWindow()
```

```
    {
```

```
        InitializeComponent();
```

```
    }
```

```
private void Window_Loaded(object sender, RoutedEventArgs
```

```
{
```

```
    service = new AppControlService(this);
```

```
    host = new ServiceHost(service);
```

```
    host.Open();
```

```
}
```

```
private void Window_Closing(object sender,
```

```
    System.ComponentModel.CancelEventArgs e)
```

```
{
```

```
    host.Close();
```

```
}
```

```
internal void SetRadius(double radius, string foreTo,
```

```
    TimeSpan duration)
```

```
{
```

```
    if (radius > 200)
```

```
{
```

```
radius = 200;
```

```
}
```

```
Color foreToColor = Colors.Red;
```

```
try
```

```
{
```

```
foreToColor = (Color)ColorConverter.ConvertFromStrin
```



```
}
```

```
catch
```

```
{
```

```
    // Ignore color conversion failure.
```

```
}
```

```
Duration animationLength = new Duration(duration);
```

```
DoubleAnimation radiusAnimation = new DoubleAnimation(
```

```
radius * 2, animationLength);
```

```
ColorAnimation colorAnimation = new ColorAnimation(
```

```
foreToColor, animationLength);
```

```
AnimatableEllipse.BeginAnimation(Ellipse.HeightProperty,
```

```
radiusAnimation);
```

```
AnimatableEllipse.BeginAnimation(Ellipse.WidthProperty,
```

```
radiusAnimation);
```

```
((RadialGradientBrush)AnimatableEllipse.Fill).Gradients
```

```
.BeginAnimation(GradientStop.ColorProperty, colorAn:
```

```
}
```

```
}
```

```
}
```

（5）修改IAppControlService.cs中的代码，如下所示：

```
[ServiceContract]
```

```
public interface IAppControlService
```

```
{
```

```
    [OperationContract]
```

```
        void SetRadius(int radius, string foreTo, int seconds);  
  
    }  
}
```

（6）修改AppControlService.cs中的代码，如下所示：

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Singl
```

```
public class AppControlService : IAppControlService  
{  
    private MainWindow hostApp;  
  
    public AppControlService(MainWindow hostApp)  
  
    {  
    }  
}
```

```
this.hostApp = hostApp;
```

```
}
```

```
public void SetRadius(int radius, string foreTo, int secon
```

```
{
```

```
hostApp.SetRadius(radius, foreTo, new TimeSpan(0, 0, secon
```

```
}
```

```
}
```

(7) 修改app.config中的代码，如下所示：

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Ch22Ex03.AppControlService">

        <endpoint address="net.tcp://localhost:8081/AppControls

          binding="netTcpBinding"

          contract="Ch22Ex03.IAppControlService" />

        </service>

      </services>
    </system.serviceModel>
```

</configuration>

(8) 在项目中添加一个新的控制台应用程序Ch22Ex03Client。

(9) 在Solution Explorer中右击解决方案，单击Set StartUp Projects。

(10) 配置解决方案，使其有多个启动项目，让多个项目同时启动。

(11) 在Ch22Ex03Client项目中添加对System.ServiceModel.dll和Ch22Ex03的引用。

(12) 修改Program.cs中的代码，如下所示：

```
using Ch22Ex03;
using System.ServiceModel;
using static System.Console;
namespace Ch22Ex03Client
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Press enter to begin.");
            ReadLine();
            WriteLine("Opening channel.");
            IAppControlService client =
                ChannelFactory<IAppControlService>.CreateChannel(
```

```

        new NetTcpBinding(),
        new EndpointAddress(
            "net.tcp://localhost:8081/AppControlService"));
WriteLine("Creating sun.");
client.SetRadius(100, "yellow", 3);
WriteLine("Press enter to continue.");
ReadLine();
WriteLine("Growing sun to red giant.");
client.SetRadius(200, "Red", 5);
WriteLine("Press enter to continue.");
ReadLine();
WriteLine("Collapsing sun to neutron star.");
client.SetRadius(50, "AliceBlue", 2);
WriteLine("Finished. Press enter to exit.");
ReadLine();
    }
}
}

```

(13) 运行解决方案。出现提示时，打开Windows防火墙TCP端口，使WCF可以监听连接。

(14) 显示Stellar Evolution窗口和控制台应用程序窗口时，在控制台窗口中按下回车键。结果如图22-5所示。



图22-5

(15) 在控制台窗口中继续按下回车键，继续星体演化循环。

(16) 关闭Stellar Evolution窗口，停止调试。

示例的说明

该示例在WPF应用程序中添加了一个WCF服务，用它控制Ellipse控件的动画。我们创建了一个简单客户端应用程序来测试服务。如果不熟悉WPF，不必过多地考虑示例中的XAML代码，我们只对WCF感兴趣。

WCF服务AppControlService有一个操作SetRadius()，客户端程序调用这个操作来控制动画。这个方法与和它同名的方法通信，同名方法在

WPF应用程序的Window1类中定义。为此，服务必须引用应用程序，所以必须驻留该服务的一个对象实例。如前所述，这意味着服务必须使用行为特性：

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Singl
public class AppControlService : IAppControlService
{
    ...
}
```

在Window1.xaml.cs中，在Windows_Loaded()事件处理程序中创建服务实例。这个方法也为服务创建了一个ServiceHost对象，并调用了其Open()方法，以便开始驻留：

```
public partial class Window1 : Window
{
    private AppControlService service;
    private ServiceHost host;
    ...
    private void Window_Loaded(object sender, RoutedEventArgs
    {
        service = new AppControlService(this);
        host = new ServiceHost(service);
        host.Open();
    }
}
```

在Window_Closing()事件处理程序中，应用程序关闭时，驻留过程中断。

配置文件非常简单，它定义了WCF服务的一个端点，监听端口8081的net.tcp地址，使用默认的NetTcpBinding绑定：

```
<service name="Ch22Ex03.AppControlService">
  <endpoint address="net.tcp://localhost:8081/AppControlSer
    binding="netTcpBinding"
    contract="Ch22Ex03.IAppControlService" />
</service>
```

这与客户端应用程序中的代码相匹配：

```
IAppControlService client =
  ChannelFactory<IAppControlService>.CreateChannel(
    new NetTcpBinding(),
    new EndpointAddress(
      "net.tcp://localhost:8081/AppControlService"));
```

客户端程序创建客户代理类时，可以调用SetRadius()方法，给它传递半径、颜色和动画持续时间等参数，这些都会通过服务转发给WPF应用程序。接着，WPF应用程序中的简单代码定义并使用动画，来改变椭圆的大小和颜色。

如果使用一个计算机名，而不是localhost，并且网络允许在指定的端口上通信，这段代码就可以在网络上工作。另外，还可以进一步分离客户端程序和宿主应用程序，并通过Internet连接起来。无论采用什么方式，WCF服务都提供了很好的通信方式，建立这种通信不需要付出过多努力。

22.4 练习

(1) 下面哪些应用程序可以驻留WCF服务？

- a. Web应用程序
- b. Windows窗体应用程序
- c. Windows服务
- d. COM+应用程序
- e. 控制台应用程序

(2) 如果要与WCF服务交换MyClass类型的参数，应实现什么类型的协定？需要什么特性？

(3) 如果把WCF服务驻留在Web应用程序中，应对服务使用的基端点进行什么扩展？

(4) 在自驻留WCF服务时，必须设置ServiceHost类的属性，调用它的方法，来配置服务。对吗？

(5) 提供服务协定IMusicPlayer的代码，它定义了Play()、Stop()和GetTrackInformation()操作。在合适的地方使用单向方法。还要为这个服务定义其他什么协定？

附录A给出了练习答案。

22.5 本章要点

第23章 通用应用程序

本章内容：

- 支持Windows 10设备进行开发
- 使用XAML和C#开发Windows通用应用程序
- 使用常见的Windows通用应用程序
- 打包和部署应用程序

本章源代码下载：

本章源代码的下载地址为

www.wrox.com/go/beginningvisualc#2015programming。从该网页的Download Code选项卡下载Chapter 23 Code后，可以找到与本章示例对应的单独文件。

Windows通用应用程序是世界各地的Windows开发人员的一个热门话题。Microsoft发布Windows 8版本是一个巨大的飞跃，从只针对台式机和笔记本电脑，变成针对平板电脑和智能手机的一个真正的市场弄潮儿。Windows 8附带一个新的API，来开发应用程序和Windows Store，允许用户采用安全、可预测的方法下载应用程序。在Windows 10和通用Windows平台（UWP）上，Microsoft引入了通用应用程序，将应用程序的效率提高到新水平。这些应用程序可以针对所有Windows平台，包括

Xbox上的手机，以及Windows桌面。

23.1 入门

编写通用应用程序之前，需要几个初始步骤。在Visual Studio旧版本中，需要得到一个Windows 8开发者许可，并应经常更新。对于Windows 10，开发不再需要该许可，但仍需要一个存储账户，以便能发布应用程序。在开发应用程序时，可以简单注册Windows 10设备，以用于开发。

在开始处理Windows通用应用程序之前，必须在设备上启用开发功能，除非它们已经安装，否则必须安装Universal Windows App Development Tools。

如果使用的是Visual Studio Express for Windows 10，或者打开一个解决方案，在Visual Studio的另一个版本中创建Windows通用应用程序，会显示如图23-1所示的对话框，提示启用Developer Mode。看到这个对话框时，单击[settings for developers](#)链接，选择Developer Mode选项，此后将看到一条警告，指出选择的选项不够安全，单击yes。

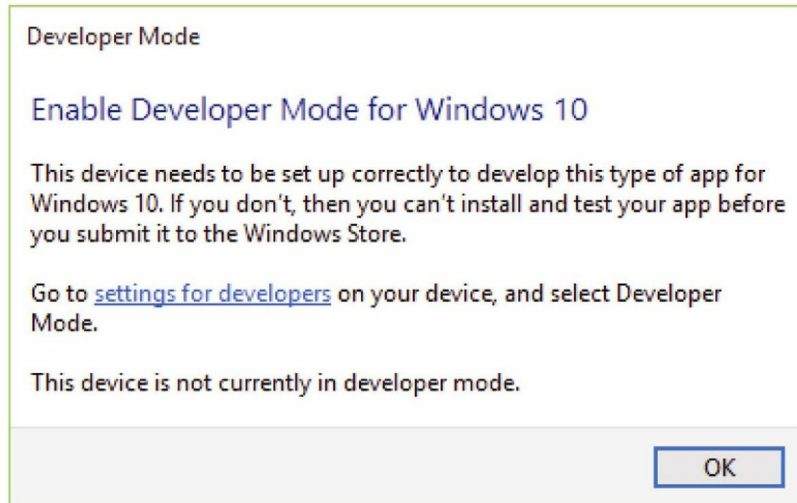


图23-1

注意： 开发者模式有两种选择：**Sideloaded**应用程序和开发者模式。**Sideloaded**应用是一种更安全的选择，因为在这个模式下，不能在设备上安装不可信的应用程序。然而，开发者模式允许在设备上调试应用程序，所以本章需要这种模式。

用户可能没有安装Universal Windows App Development Tools。一些版本的Visual Studio会自动安装它，但是如果没有安装，只需要打开New projects对话框，然后选择Visual C#[Windows|Universal，就会看到一个链接。单击这个链接，安装工具。

23.2 通用应用程序

Windows通用应用程序可以针对多个设备类型。传统的应用程序，像本书前面编写的WPF桌面游戏，针对单一设备类型，比如PC。Microsoft引入了通用Windows平台，可以编写出能够运行在多个设备上的单个应用程序，并已投入了许多精力，提升开发者开发这种应用程序的体验。

在大量异构设备上开发应用程序的主要挑战是，无法提前得知屏幕有多大，或用户将如何与设备交互。如果只考虑把本书前面的Karli Card WPF应用程序放在手机屏幕上，哪怕屏幕最大的手机，该应用程序看起来也很可怕。另一个方面是手机用户希望应用程序能够调整它在屏幕上的显示方向。本章将介绍响应UI和适应性触发器的概念，来解决这些问题。

通用应用程序通过Windows Store部署，对于打包应用程序而言，这有其自身的挑战。为将应用程序放在Windows Store上，必须经历一个相当严格的测试过程，通过Microsoft设定的许多要求。本章最后将讨论这个过程，以便你发布自己的应用程序。

23.3 应用程序概念和设计

应用程序如何在手机和Windows桌面中显示有着极大的差异。运行在Windows桌面上的应用程序设计在很大程度上是不变的，虽然因为Windows 95的引入，这种应用程序有更好的图形。其设计特性是一个窗口带有标题栏，右上角有三个按钮用于最大化、最小化、关闭应用程序，还包含按钮、单选按钮、复选框等来显示内容。引入Windows 8后，应用程序的生成稍有不同。它们通过触摸来工作，而不是鼠标和键盘，标题栏可能有，也可能没有，可以旋转，以适应运行它们的设备的方向，这只是几个差异。

Microsoft推出Windows 8时，还发布了一个相当大的应用设计指南，即使不必坚持使用Windows 8，也应该知道有这个指南。尽管应用程序运行在各种设备上，但它们有许多共同的特征。所以下面介绍一些共同特征，看看Windows Store应用程序如何和桌面应用程序匹配。

注意：Windows 8应用程序设计指南可以在<http://go.microsoft.com/fwlink/p/?linkid=258743>下载。

23.3.1 屏幕方向

所有Windows应用程序都应能优雅地调整自己的大小。特别重要的

一个方面是手持设备可以在三维空间中移动。用户会期待应用程序随着屏幕的方向来移动。因此，如果用户倒转平板电脑，应用程序应该随之倒转。

23.3.2 菜单和工具栏

经典桌面应用程序使用菜单和工具栏在视图之间导航。通用应用程序也可以这样做，但它们更有可能使用工具栏，而不是菜单。桌面应用程序通常总是显示菜单和工具栏的可视化组件，但是通用应用程序往往会选择不这样做，以在较小的屏幕上节省宝贵的空间。

不是强迫用户通过菜单发现应用程序的复杂性，应用程序风格把应用程序呈现给用户，他们可以在需要的时候激活菜单。当菜单显示出来时，应该很简单，只包含主选项。由用户来决定何时何地显示菜单。

23.3.3 磁贴和徽章

Windows使用活动磁贴（Tile）在开始菜单和页面上显示应用程序。该名称中的“活动”源于如下事实：磁贴可以基于应用程序的当前内容或状态而改变。例如，照片应用程序会旋转开始页面上的照片，邮件客户端显示未读邮件的数量，游戏显示上次保存的截图等。这种可能性几乎是无止境的。

为应用程序提供好的磁贴比为应用程序桌面提供好的图标更重要，这非常重要。磁贴嵌在应用程序的清单里，参见本章稍后的内容，使用Visual Studio很容易包括它们。

徽章（badge）是磁贴的一个小版本，Windows可在锁定屏幕和其他情况下使用它。不需要为应用程序提供徽章，除非要在锁定屏幕上显示通知。

23.3.4 应用程序的生存期

经典的Windows桌面应用程序可以通过单击标题栏右上角的一个按钮来关闭，但通用应用程序通常不显示标题栏，那么该如何关闭它？一般来说，不需要关闭通用应用程序。只要通用应用程序失去焦点，就会挂起，并完全停止使用处理器资源。这就允许许多应用程序同时运行，而事实上它们只是暂停。在Windows中，应用程序失去焦点，就会自动暂停。用户并未真正注意到，但应用程序开发人员应该认识到这个非常重要的事实，并处理它。

23.3.5 锁屏应用程序

一些应用程序失去焦点时应该继续运行。这种应用程序的示例包括GPS导航和音频流应用。即使用户开始开车或开始使用其他应用程序，也希望这类应用程序继续运行。如果应用程序需要继续在后台运行，就必须把它声明为锁定屏幕应用程序，并提供信息，以便在锁定屏幕上显示通知。

23.4 应用程序的开发

开始开发Windows通用应用程序时，有很多关于编程和UI语言的选择。本书使用C#和XAML，其他选项包括使用JavaScript和HTML5、C++和DirectX或Visual Basic和XAML。

用于创建通用应用程序用户界面的XAML，与WPF使用的XAML并不完全相同，但它们足够接近，使用起来应感到得心应手。很多熟悉的控件也存在于通用应用程序，但它们的外观与Windows桌面的控件略有不同，还有许多为触摸进行了优化的控件。

注意：与Windows 8应用程序一样，Microsoft已经发布了通用应用程序的设计指南。该指南可在
<https://msdn.microsoft.com/library/windows/apps/hh465424.aspx>找到。

23.4.1 自适应显示

自适应显示指显示的内容能响应用户行为的变化，如手机翻到一侧，或者窗口改变大小。当用户翻转手机时，应用程序应能够优雅地从纵向模式切换到横向模式，无论应用程序部署到笔记本电脑还是手机上，都应能工作。

创建新的Windows通用应用项目时，首先会注意到，在设计器中显

示的页面看起来很小。这是因为这个项目默认为使用为5英寸手机显示屏进行优化的视图。可以使用如图23-2所示的Device Preview面板改变这个设置。还可以使用这个面板把纵向布局改为横向。

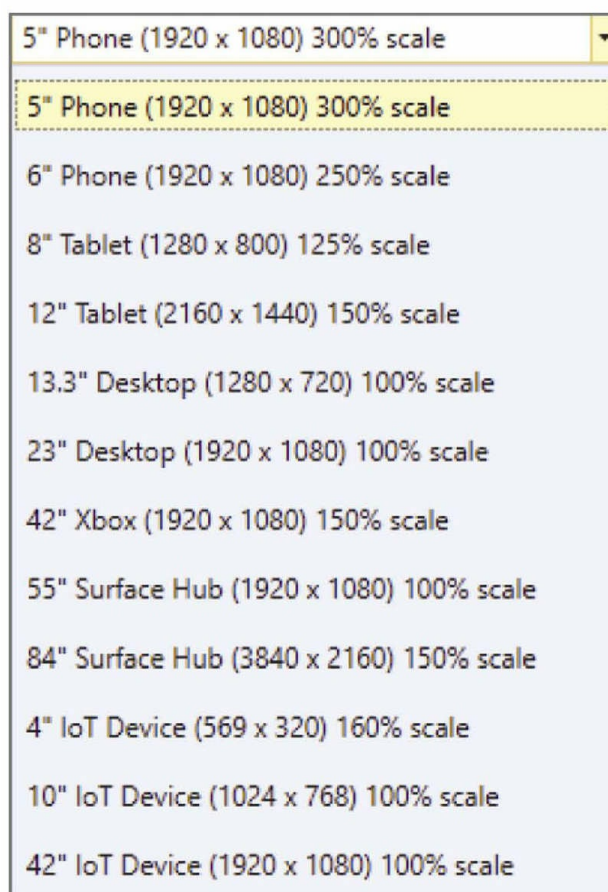


图23-2

行为良好的应用程序能在Device Preview面板显示的许多（但不是所有）窗体元素中显示出来。考虑到这个列表的范围是物联网（Internet of Things, IoT）设备的569×320像素到Surface Hub的3840×2160像素，这是一个艰巨的任务。幸好，Visual Studio和通用Windows平台框架会提供帮助。从下拉框中改变分辨率（或屏幕大小）时，Visual Studio将调整应用程序的大小，用户马上就能看到页面是什么样子。此外，帮助

给应用程序创建自适应设计的控件，都包含在工具箱中，可以利用它们轻松创建易于变换的UI。

1. 相对面板

在第14和第15章中，使用Grid和StackPanels控件创建一个UI，它能提供很好的静态显示。但是在这个世界上，必须面对很多显示屏尺寸，所以必须有某种东西可以更好地移动控件。这就是RelativePanel控件。

相对面板允许指定一个控件应该如何相对于另一个控件来定位。可以把控件相放在其他控件的左边、右边、上边或下边，也可以完成其他一些很好的处理。可以相对于一个控件的左、右或中心来水平和垂直放置另一个控件，使控件的边缘与面板的边缘对齐。这意味着不再需要利用像素让两个控件在显示屏上完美对齐。

2. 自适应触发器

自适应触发器是Visual State Manager的新增功能。使用这些触发器可以基于显示屏的大小更改应用程序的布局。与相对面板一起使用时，这是一个非常强大的功能，可以用相当简单的方式构建网络世界所谓的响应性UI，Microsoft称之为自适应显示。

试一试：自适应显示：**Ch23Ex01**

(1) 选择File|New|Project，展开Installed|Visual C#|Windows|Universal，创建一个新的Windows通用应用项目，选择Blank App（Universal Windows）项目，并命名为AdaptiveDisplay。

(2) 把一个RelativePanel控件添加到网格中。其边距设置为20，将HorizontalAlignment设置为Stretch。

(3) 在面板上添加textBlock和文本框：

```
<RelativePanel HorizontalAlignment="Stretch" Margin="20" >
    <TextBlock x:Name="textBlockFirstName" Text="First name"
    <TextBox x:Name="textBoxFirstName" Text="" Width="400"
RelativePanel.RightOf="textBlockFirstName"
RelativePanel.AlignVerticalCenterWith="textBlockFirstName" />
</RelativePanel>
```

(4) 在网格上添加一个Visual State Manager。这很重要，因为它是网格的第一个子元素。

```
<VisualStateManager.VisualStateGroups>
<VisualStateGroup>
    <VisualState x:Name="narrowView">
        <VisualState.StateTriggers>
            <AdaptiveTrigger MinWindowWidth="0" />
        </VisualState.StateTriggers>
        <VisualState.Setters>
            <Setter Target="textBoxFirstName.(RelativePanel.Below)"
                Value="textBlockFirstName" />
        </VisualState.Setters>
    </VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

```

        <Setter
            Target="textBoxFirstName.(RelativePanel.AlignVer
                Value="" />
        <Setter
            Target="textBoxFirstName.(RelativePanel.Align
                Value="textBlockFirstName" />
    </VisualState.Setters>
</VisualState>
    <VisualState x:Name="wideView">
        <VisualState.StateTriggers>
            <AdaptiveTrigger MinWindowWidth="720" />
        </VisualState.StateTriggers>
    </VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>

```

(5) 在Device Preview下拉框中更改目标显示设备。选择一个较小的手机显示屏时，在TextBlock下面就会弹出文本框。如果选择平板电脑或另一个更大的显示器，在TextBlock的右边就会弹出文本框。

示例的说明

这个例子中的Visual State Manager是根网格的第一个子元素，这很重要，它允许解释器找到要引用的控件。如果把它放在另一个位置，不会出错，但也不会得到预期的结果。

Visual State Manager使用AdaptiveTrigger和MinWindowWidth属性来

改变显示器的行为：

```
<AdaptiveTrigger MinWindowWidth="0" />
```

我们定义了两个状态，如果视图至少0像素宽，就激活其中一个，如果视图至少720像素宽，就激活另一个。你可能会认为，视图的宽度超过720像素时，两个状态都会激活，但它不是这样工作的。相反，在任何时候都只激活一个状态，并选择最匹配的那个状态。所以，当视图是1024像素宽时，就仅选中wide状态。

在narrowView中，设置了三个属性：

```
<VisualState.Setters>
    <Setter Target="textBoxFirstName.(RelativePanel.Below
        Value="textBlockFirstName" />
    <Setter
        Target="textBoxFirstName.(RelativePanel.Ali
        Value="" />
    <Setter
        Target="textBoxFirstName.(RelativePanel.Ali
        Value="textBlockFirstName" />
```

首先，确保文本框移到TextBlock的下面。第二，清除AlignVerticalCenterWith属性。如果不改变它，将否决把控件移到TextBlock下面的指令。这是因为AlignVerticalCenterWith属性直接在控件上设置，如果不清除它，它将优先于View State的Below指令。另一种方法是避免直接在控件上设置的任何属性，只使用视图状态。最后，对齐控件的左边缘。

`wideView`状态实际上是空的。这意味着不应修改直接在控件上定义的属性，而应使之处于默认状态。

注意：当前版本的Visual Studio有时无法基于Device Preview面板上的选择移动控件。如果发生这种情况，应从下拉框中选择另一个视图大小，视图应调整正确。

3. FlipView

`FlipView`是个不错的小控件，非常适合于手持设备。它允许用户向左或向右滑动屏幕，来显示一些内容。它通常用于一次显示一张图像，允许用户使用滑动手势在图像之间移动。

默认情况下，`FlipView`允许用户向左或向右移动视图中的内容，但也可以改为向上或向下移动。使用鼠标时，滚动按钮也有效。

试一试：FlipView: Ch23Ex02

(1) 选择File|New|Project，展开Installed|Visual C#|Windows|Universal，创建一个新的Windows通用应用项目，选择Blank App（Universal Windows）项目，并命名为PictureViewer。

(2) 把三个RelativePanel控件添加到MainPage的Grid选项卡上。

```

<RelativePanel Margin="20">
    <RelativePanel x:Name="LeftPanel" Margin="0,10,0,0" >
    </RelativePanel>
    <RelativePanel x:Name="RightPanel" Margin="20,10,0,0">
    </RelativePanel>
</RelativePanel>

```

(3) 把一个FlipView添加到LeftPanel面板上，如下所示：

```

<FlipView x:Name="flipView" VerticalAlignment="Stretch" HorizontalAlignment="Stretch">
    <FlipView.ItemTemplate>
        <DataTemplate>
            <Image x:Name="image" Source="{Binding}" Stretch="UniformToFill" />
        </DataTemplate>
    </FlipView.ItemTemplate>
</FlipView>

```

(4) 把3个TextBlock添加到RightPanel面板上，如下所示：

```

<TextBlock x:Name="textBlockCurrentImageDisplayName"
    Margin="10,10,10,0" FontSize="24" FontWeight="Bold"
    RelativePanel.AlignLeftWithPanel="True"
    RelativePanel.AlignRightWithPanel="True" />
<TextBlock x:Name="textBlockCurrentImageImageHeight" Margin="10,10,10,0"
    FontSize="24" FontWeight="Bold"
    RelativePanel.AlignLeftWithPanel="True"
    RelativePanel.AlignRightWithPanel="True"
    RelativePanel.Below="textBlockCurrentImageDisplayName" />

```

```

<TextBlock x:Name="textBlockCurrentImageImageWidth" Margin:
    FontSize="24" FontWeight="Bold"
    RelativePanel.AlignLeftWithPanel="True"
    RelativePanel.AlignRightWithPanel="True"
    RelativePanel.Below="textBlockCurrentImageImageH

```

(5) 给控件添加以下Visual State Manager，在应用程序调整大小来控制其外观。把它添加为Grid标记的第一个子元素：

```

<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="narrowView">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0" />
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="RightPanel.(RelativePanel.Below)" Value="textBlockCurrentImageImageH" />
        <Setter Target="RightPanel.(RelativePanel.AlignLeftWithPanel)" Value="True" />
        <Setter Target="RightPanel.(RelativePanel.AlignRightWithPanel)" Value="True" />
        <Setter Target="RightPanel.Margin" Value="0,10,0,0" />
        <Setter Target="LeftPanel.(RelativePanel.AlignTopWithPanel)" Value="True" />
        <Setter Target="LeftPanel.(RelativePanel.AlignLeftWithPanel)" Value="True" />
        <Setter Target="LeftPanel.(RelativePanel.AlignRightWithPanel)" Value="True" />
        <Setter Target="LeftPanel.Height" Value="560" />
      </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="wideView">

```

```

<VisualState.StateTriggers>
    <AdaptiveTrigger MinWindowWidth="720" />
</VisualState.StateTriggers>
<VisualState.Setters>
    <Setter Target="RightPanel.(RelativePanel.AlignBottom" Value="Stretch" />
    <Setter Target="RightPanel.(RelativePanel.AlignRight" Value="Stretch" />
    <Setter Target="RightPanel.(RelativePanel.AlignTopWithParent" Value="Stretch" />
    <Setter Target="RightPanel.Width" Value="200" />
    <Setter Target="LeftPanel.(RelativePanel.LeftOf)" Value="RightPanel" />
    <Setter Target="LeftPanel.(RelativePanel.AlignBottom" Value="Stretch" />
    <Setter Target="LeftPanel.(RelativePanel.AlignTopWithParent" Value="Stretch" />
    <Setter Target="LeftPanel.(RelativePanel.AlignLeftWithParent" Value="Stretch" />
</VisualState.Setters>
</VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>

```

(6) 创建一个新的类，命名为ImageProperties。添加三个属性，如下所示：

```

namespace PictureBox
{
    class ImageProperties
    {
        public string FileName { get; set; }
        public int Width { get; set; }
        public int Height { get; set; }
    }
}

```

```
}  
}
```

(7) 进入主页的后台隐藏代码，添加如下using语句：

```
using System;  
using System.Collections.Generic;  
using Windows.Storage;  
using Windows.UI.Xaml;  
using Windows.UI.Xaml.Controls;  
using Windows.Storage.Search;  
using Windows.UI.Xaml.Media.Imaging;  
using Windows.UI.Popups;  
using System.Linq;
```

(8) 创建一个私有字段，来保存要显示图片的一些信息：

```
private IList<ImageProperties> imageProperties = new List<Image
```

(9) 添加一个方法，来加载文件：

```
private async void GetFiles()  
{  
    try  
    {  
        StorageFolder picturesFolder = KnownFolders.PicturesLibr  
        IReadOnlyList<StorageFile> sortedItems = await  
picturesFolder.GetFilesAsync(CommonFileQuery.OrderByDate);  
        var images = new List<BitmapImage>();
```



```

if (sortedItems.Any())
{
    foreach (StorageFile file in sortedItems)
    {
        if (file.FileType.ToUpper() == ".JPG")
        {
            using (Windows.Storage.Streams.IRandomAccessStream
file.OpenAsync(FileAccessMode.Read))
            {
                BitmapImage bitmapImage = new BitmapImage();
                await bitmapImage.SetSourceAsync(fileStream);
                images.Add(bitmapImage);
                imageProperties.Add(new ImageProperties
                {
                    FileName = file.DisplayName,
                    Height = bitmapImage.PixelHeight,
                    Width = bitmapImage.PixelWidth
                });
                if (imageProperties.Count > 10)
                    break;
            }
        }
    }
}
else
{
    var message = new MessageDialog("There are no images i

```

```

        await message.ShowAsync();
    }
    flipView.ItemsSource = images;
}
catch (UnauthorizedAccessException)
{
    var message = new MessageDialog("The app does not have a
on this device.");
    await message.ShowAsync();
}
}

```

(10) 在主页面的XAML中选择Page标记，添加Loading事件。然后实现这个事件的处理程序：

```

private void Page_Loaded(object sender, RoutedEventArgs e)
{
    GetFiles();
}

```

(11) 在主页面的XAML中选择FlipView，实现SelectionChanged事件：

```

private void flipView_SelectionChanged(object sender, Selecti
{
    if (flipView.SelectedIndex >= 0)
    {
        textBlockCurrentImageDisplayName.Text =

```

```

imageProperties[flipView.SelectedIndex].FileName;
        textBlockCurrentImageImageHeight.Text =
imageProperties[flipView.SelectedIndex].Height.ToString();
        textBlockCurrentImageImageWidth.Text =
imageProperties[flipView.SelectedIndex].Width.ToString();
    }
}

```

(12) 最后双击Package.appxmanifest文件，打开清单文件设计器。

(13) 选择Capabilities选项卡，确保选中Pictures Library功能。

(14) 运行该应用程序。

示例的说明

代码使用三个RelativePanel来移动它的内容。所有面板都没有直接的定位指令，整个布局都是在Visual State Manager中定义的。本例使用了两个自适应触发器，如果视图宽于720像素，就激活一个自适应触发器，如果视图宽于0像素，就激活另一个自适应触发器。

FlipView的代码在本例中最少。

```

<FlipView x:Name="flipView" VerticalAlignment="Stretch"
HorizontalAlignment="Stretch" >
    <FlipView.ItemTemplate>
        <DataTemplate>
            <Image x:Name="image" Source="{Binding}" Stretch="

```

```
</DataTemplate>
</FlipView.ItemTemplate>
</FlipView>
```

这段代码告诉FlipView，应该使用这里定义的ItemTemplate，它只包括一个Image控件。很明显，可以使用FlipView显示任何内容，而不仅是图片。

Getfile方法中的代码演示了几个接口，它们可以用于访问应用程序中的文件和资源。本章后面将讨论沙箱应用程序的概念，以及它们对代码的限制，但是本例已经展示了这个概念的一些方面。如果应用程序有权访问StorageFolder对象，下面的代码就会获得它，否则就抛出UnauthorizedAccessException异常。

```
StorageFolder picturesFolder = KnownFolders.PicturesLibrary;
```

在正常.NET中，没有这个类，要根据用户在文件系统中的权限，来确定是否授予访问权限。对于应用程序来说，这是非常不同的。这里必须事先声明应用程序需要访问哪些资源，用户只有接受这些权限，应用程序才能运行。在步骤13中，声明应用将包括访问照片库的能力。如果不这样做，运行应用程序时会出现异常。

接下来使用StorageFolder的GetFilesAsync方法检索文件，并按日期排列。

一旦有了文件，就调用StorageFile对象上的OpenAsync，打开它们。

```
using (Windows.Storage.Streams.IRandomAccessStream fileStream
```

```
file.OpenAsync(FileAccessMode.Read))
```

这返回一个文件流，它可以用于访问文件的内容。本例不希望写入内容，所以只指定了Read访问。

最后，把FlipView的ItemsSource设置为从图片库中加载的图片列表。

23.4.2 沙箱应用程序

现在应回过头来，看看Windows通用平台的.NET框架的局限性。运行在移动设备的应用程序对运行它们的操作系统只有有限的访问权限，这意味着不能编写某些类型的应用程序。如果需要直接访问文件系统，以访问Windows系统文件，就必须编写经典的Windows桌面应用程序。

在C#中编写通用应用程序时，会发现限制因素在应用程序引用的.NET Framework中，其中常见的名称空间和类完全缺失，或可用的方法比以前更少。如果打开Visual Studio，创建一个新的Blank应用程序，然后扩展References节点，将看到该引用非常不同于Windows桌面应用程序。有三个ApplicationInsights引用，每个都允许监控应用程序的各个方面，还有两个对.NET和Windows的引用。对.NET的引用是.NET的一个修订版本，对Windows的引用是Windows Core API。在这一点上，你可能会认为，可以简单地改变引用，使用正常的.NET Framework。事实上这是有效的。也就是说，它此时会正常工作，但当用户试着把应用程序发布到Windows Store时，会因为与规范不兼容而拒绝它。

Windows通用应用程序的沙箱性质，以及它们获得Windows Store认

可之前必须经历的过程，意味着用户应该很少担心通过Store会下载到恶意的应用程序。显然，有些人会试图规避这一点，用户不应该放松警惕；然而，通过Windows Store把恶意程序放在Windows计算机上，要大大难于通过正常方式来下载和安装应用程序。

1. 磁盘访问

桌面应用程序差不多可以随意访问磁盘，但有一些例外。一个这样的例外是，通常禁止它们写入Program Files文件夹和其他系统文件夹。Windows通用应用程序只能直接访问少数非常特定的磁盘位置。这些位置包括安装应用程序的文件夹、与应用程序相关的AppData文件夹以及一些特殊文件夹，如“文档”文件夹。文件和文件夹的访问权限也移到通用应用程序的.NET Framework中，确保开发人员不会意外地写入某个被禁止的位置。

为允许用户控制应该在应用程序中存储和读取什么地方的文件，Windows提供了三个File Picker合同：FolderOpenPicker、FileOpenPicker和FileSavePicker。这些选择器类可以在应用程序中用于获得本地磁盘的安全访问权限。

如前所述，也可以使用KnownFolders类访问设备上的资源。对于要读写的位置，如果只有用户拥有访问权限，应用程序才能打开它们，则应使用KnownFolders类。

2. 串行化、流和异步编程

第14章使用[Serializable]特性允许类的序列化。通用应用程序的.NET不包含这个特性，但可以使用一个类似的特性[DataContract]。DataContract特性使用DataContractSerializer类来序列化类的内容。为把序列化的内容放在磁盘上或从磁盘上序列化，需要使用一些文件访问类型，但与正常.NET不同，不能直接创建它们。而应使用文件选择器创建流对象，再用流对象和DataContractSerializer来保存、加载文件。

注意： 可以从
www.wrox.com/go/beginningvisualc#2015programming下载的本章项目包括一个你可能无法使用的证书文件，但可以自己生成它。遵循以下步骤：

- (1) 打开项目，双击Package.appxmanifest文件。
- (2) 选择Packaging选项卡。
- (3) 点击Choose Certificate。
- (4) 从Configure Certificate中选择Create test certificate。
- (5) 单击OK。

下一个例子演示了借助DataContractSerializator，并结合使用通过FileOpenPicker、FileSavePicker创建的流，来加载和保存数据模型的XML表示。

试一试：磁盘访问：Ch23Ex03

(1) 在Visual Studio中选择Blank App (Universal Windows)，创建一个新项目，命名为DataSerialization。

(2) 在项目中创建一个新类，命名为AppData。

(3) 用[DataContract]特性标记类，把System.Runtime.Serialization名称空间添加到using部分：

```
using System.Runtime.Serialization;
namespace DataSerialization
{
    [DataContract]
    class AppData
    {
    }
}
```

(4) 给类添加一个int类型的属性，用[DataMember]特性标记它：

```
[DataMember]
public int TheAnswer { get; set; }
```

(5) 给项目添加一个新的枚举AppStates。用[DataContract]特性标记它：


```

using System.Runtime.Serialization;
namespace DataSerialization
{
    [DataContract]
    public enum AppStates
    {
    }
}

```

(6) 给AppStates添加三个值，用[EnumMember]特性分别进行标记：

```

[EnumMember]
Started,
[EnumMember]
Suspended,
[EnumMember]
Closing

```

(7) 给AppData类添加两个新属性：

```

[DataMember]
public AppStates State { get; set; }
[DataMember]
public object StateData { get; set; }

```

(8) 添加一个新类AppStateData，用[DataContract]特性标记它：

```

using System.Runtime.Serialization;

```

```

namespace DataSerialization
{
    [DataContract]
    public class AppStateData
    {
        [DataMember]
        public string Data { get; set; }
    }
}

```

(9) 给AppData类添加[KnownType]特性，如下：

```

[DataContract]
[KnownType(typeof(AppStateData))]
public class AppData
{

```

(10) 在Solution Explorer中双击MainPage.xaml文件，把两个按钮拖到页面中。把其内容和名称属性设置为Save和Load。

(11) 为Save按钮创建一个单击事件处理程序，在代码隐藏文件中导航到它。添加如下代码（注意方法声明中的async关键字）：

```

private async void Save_Click(object sender, RoutedEventArgs)
{
    var data = new AppData
    {
        State = AppStates.Started,

```

```

        TheAnswer = 42,
        StateData = new AppStateData { Data = "The data is b
    };
    var fileSavePicker = new FileSavePicker
    {
        SuggestedStartLocation = PickerLocationId.DocumentsLibr
        DefaultFileExtension = ".xml",
    };
    fileSavePicker.FileTypeChoices.Add("XML file", new[] {
var file = await fileSavePicker.PickSaveFileAsync();
if (file != null)
{
    var stream = await file.OpenStreamForWriteAsync();
    var serializer = new DataContractSerializer(typeof(App
    serializer.WriteObject(stream, data);
}
}

```

(12) 创建Load按钮的单击事件处理程序，添加如下代码（再次注意async关键字）：

```

private async void Load_Click(object sender, RoutedEventArgs)
{
    var fileOpenPicker = new FileOpenPicker
    {
        SuggestedStartLocation = PickerLocationId.DocumentsLibr
        ViewMode = PickerViewMode.Thumbnail
    }
}

```

```

};
fileOpenPicker.FileTypeFilter.Add(".xml");
var file = await fileOpenPicker.PickSingleFileAsync();
if (file != null)
{
    var stream = await file.OpenStreamForReadAsync();
    var serializer = new DataContractSerializer(typeof(App
    var data = serializer.ReadObject(stream);
}
}

```

(13) 需要把下面两个名称空间添加到代码隐藏文件中：

```

using System.Runtime.Serialization;
using Windows.Storage.Pickers;

```

(14) 运行该应用程序。

示例的说明

在步骤1至步骤9，创建了应用程序的数据模型。所有类和枚举都用[DataContract]特性标记，但注意成员的标记方式之间的区别。类中的属性和字段可以用[DataMember]特性标记，但枚举的成员必须用[EnumMember]特性标记：

```

[DataContract]
public class AppStateData
{

```

```

    [DataMember]
    public string Data { get; set; }
}
[DataContract]
public enum AppStates
{
    [EnumMember]
    Started,
    [EnumMember]
    Suspended,
    [EnumMember]
    Closing
}

```

这里没有显示的另一个特性`CollectionDataContract`比较有趣。它可以在自定义集合上设置。

还添加了一个带有`type`对象的属性。为了使序列化器能够序列化这个属性，必须告诉它是什么类型。为此，可以在包含该属性的类上设置`[KnownTypes]`特性。

`Save`和`Load`方法展示一些新的文件选择器。显示选择器后，就返回一个`StorageFile`实例：

```

var file = await fileOpenPicker.PickSingleFileAsync();
if (file != null)
{
    var stream = await file.OpenStreamForReadAsync();
}

```

```
var serializer = new DataContractSerializer(typeof(AppDa  
var data = serializer.ReadObject(stream);  
}
```

这个对象可以用来打开一个流，用于读写操作。这里没有直接显示，但也可以将它直接用于FileIO类，它提供了一些简单的方法，来写入和读取数据。

23.4.3 在页面之间导航

应用程序内部页面之间的导航类似于Web应用程序的导航方式。可以调用Navigate方法从一个页面导航到另一个页面，调用Back方法可以返回。下面的示例演示了如何使用三种基本页面在应用程序的页面之间移动。

试一试：导航： **Ch23Ex04**

(1) 在Visual Studio中选择Blank App (Universal Windows)，创建一个新项目，命名为BasicNavigation。

(2) 选择并删除MainPage.xaml文件。

(3) 右击该项目，并选择Add|New item。使用Blank Page模板添加一个新页面，并命名为BlankPage1。

(4) 重复步骤3两次，项目就有了三个页面，分别命名为BlankPage2和BlankPage3。

(5) 打开App.xaml.cs代码隐藏文件，定位OnLaunched方法。该方法使用刚才删除的MainPage，所以把引用改为BlankPage1。

(6) 在BlankPage1上，给网格插入一个堆栈面板、一个TextBlock和三个按钮：

```
<Grid Background="{ThemeResource ApplicationPageBackgroundT
<TextBlock x:Name="textBlockCaption" Text="Page 1" Horizontal
Margin="10" VerticalAlignment="Top"/>
<StackPanel Orientation="Horizontal" Grid.Row="1" Horizontal
    <Button Content="Page 2" Click="buttonGoto2_Click" />
    <Button Content="Page 3" Click="buttonGoto3_Click" />
    <Button Content="Back" Click="buttonGoBack_Click" />
</StackPanel>
</Grid>
```

(7) 添加单击事件的事件处理程序，如下：

```
private void buttonGoto2_Click(object sender, RoutedEventArgs)
{
    Frame.Navigate(typeof(BlankPage2));
}
private void buttonGoto3_Click(object sender, RoutedEventArgs)
{
    Frame.Navigate(typeof(BlankPage3));
}
```

```

}
private void buttonGoBack_Click(object sender, RoutedEventArgs)
{
    if (Frame.CanGoBack) this.Frame.GoBack();
}

```

(8) 打开第二页 (BlankPage2)，添加一个类似的堆栈面板：

```

<TextBlock x:Name="textBlockCaption" Text="Page 2" Horizontal
Margin="10" VerticalAlignment="Top"/>
<StackPanel Orientation="Horizontal" Grid.Row="1" Horizontal
    <Button Content="Page 1" Click="buttonGoto1_Click" />
    <Button Content="Page 3" Click="buttonGoto3_Click" />
    <Button Content="Back" Click="buttonGoBack_Click" />
</StackPanel>

```

(9) 把导航添加到事件处理程序中：

```

private void buttonGoto1_Click(object sender, RoutedEventArgs)
{
    Frame.Navigate(typeof(BlankPage1));
}
private void buttonGoto3_Click(object sender, RoutedEventArgs)
{
    Frame.Navigate(typeof(BlankPage3));
}
private void buttonGoBack_Click(object sender, RoutedEventArgs)
{

```



```

    if (Frame.CanGoBack) this.Frame.GoBack();
}

```

(10) 打开第三页，添加另一个堆栈面板，其中包括一个Home按钮：

```

<TextBlock x:Name="textBlockCaption" Text="Page 3" Horizontal
Margin="10" VerticalAlignment="Top"/>
<StackPanel Orientation="Horizontal" Grid.Row="1" Horizontal
<Button Content="Page 1" Click="buttonGoto1_Click" />
<Button Content="Page 2" Click="buttonGoto2_Click" />
<Button Content="Back" Click="buttonGoBack_Click" />
</StackPanel>

```

(11) 添加事件处理程序：

```

private void buttonGoto1_Click(object sender, RoutedEventArgs
{
    Frame.Navigate(typeof(BlankPage1));
}
private void buttonGoto2_Click(object sender, RoutedEventArgs
{
    Frame.Navigate(typeof(BlankPage2));
}
private void buttonGoBack_Click(object sender, RoutedEventArgs
{
    if (Frame.CanGoBack) this.Frame.GoBack();
}

```

(12) 运行应用程序。该应用程序显示有三个按钮的首页。

示例的说明

运行应用程序时，它显示了一个加载时的闪屏，接着显示第一个页面。第一次点击一个按钮时，使用想浏览的页面类型调用Navigate方法：

```
Frame.Navigate(typeof(BlankPage2));
```

它没有显示在这个例子中，但Navigate方法包括一个重载版本，允许把参数发送给要导航到的页面上。在页面之间导航时，请注意，如果使用一个按钮返回第一页，Back按钮仍然是活动的。

在每个页面上，使用GoBack事件的实现代码回到上一页。调用GoBack方法之前，会检查CanGoBack属性。否则，在显示的第一页上调用GoBack时，会得到一个异常。

```
if (Frame.CanGoBack) this.Frame.GoBack();
```

每次导航到一个页面，都会创建一个新实例。为了改变这一行为，可以在页面的构造函数中启用属性NavigationCacheMode，例如：

```
public BasicPage1()
{
    this.InitializeComponent();
    NavigationCacheMode = Windows.UI.Xaml.Navigation.Navigation
}
```

这会缓存页面。

23.4.4 CommandBar控件

CommandBar提供的功能与桌面应用程序的工具栏基本相同，但应该让它们简单得多，通常限制工具栏上可用的选项少于8项。

一次可以显示多个CommandBar，但请记住，这会使用户界面很杂乱，不应该只为显示更多选项而显示多个工具栏。另一方面，如果想提供多种导航，有时同时把工具栏显示顶部和底部会比较好。

Visual Studio附带了CommandBar控件，很容易创建这种类型的控件。下面的示例创建一个应用程序的工具栏，其中包含许多标准项。

试一试：创建**CommandBar**： Ch23Ex05

(1) 回到先前的BasicNavigation例子。

(2) 在三个页面上都添加一个CommandBar。把它作为每个页面上网格控件的子元素：

```
<CommandBar>
    <AppBarToggleButton x:Name="toggleButtonBold" Icon="Bold" I
Click="AppBarToggleButtonBold_Click" />
    <AppBarSeparator />
    <AppBarButton Icon="Back" Label="Back" Click="buttonGoBacI
```

```

<AppBarButton Icon="Forward" Label="Forward" Click="AppBarF
    <CommandBar.SecondaryCommands>
        <AppBarButton Icon="Camera" Label="Take picture" />
        <AppBarButton Icon="Help" Label="Help" />
    </CommandBar.SecondaryCommands>
</CommandBar>

```

(3) 把下面的事件处理程序添加到所有三个页面上:

```

private void AppBarButtonForward_Click(object sender, RoutedEventArgs)
{
    if (Frame.CanGoForward) this.Frame.GoForward();
}
private void AppBarToggleButtonBold_Click(object sender, RoutedEventArgs)
{
    AppBarToggleButton toggleButton = sender as AppBarToggleButton;
    bool isChecked = toggleButton.IsChecked.HasValue ?
        (bool)toggleButton?.IsChecked.Value : false;
    textBlockCaption.FontWeight = isChecked ? FontWeights.Bold
}

```

(4) 把下面的using语句添加到所有页面上:

```
using Windows.UI.Text;
```

(5) 在所有三个页面上, 把文本框的margin改为10,50,10,10。

(6) 运行该应用程序。

示例的说明

运行这个应用程序时，现在可以使用命令栏按钮在曾经访问过的页面列表中来回移动。命令栏本身很容易处理。

命令栏用三种类型来建立。第一个是AppBarToggleButton。

```
<AppBarToggleButton x:Name="toggleButtonBold" Icon="Bold" Label="Bold" Click="AppBarToggleButtonBold_Click" />
```

这种类型的按钮可用来显示开启或关闭的状态。

第二种类型是AppBarButton，与任何其他按钮类似，事实上可以看到，AppBarButtonBack按钮的单击事件是由前面示例中ButtonBack的同一个事件处理程序处理的。

```
<AppBarButton Icon="Back" Label="Back" Click="buttonGoBack_Click" />
```

用于命令栏的第三类是AppBarSeparator。这种控件只显示命令栏中的分隔线。

最后，两个按钮位于CommandBar.SecondaryCommands标签内：

```
<CommandBar.SecondaryCommands>
    <AppBarButton Icon="Camera" Label="Take picture" />
    <AppBarButton Icon="Help" Label="Help" />
</CommandBar.SecondaryCommands>
</CommandBar>
```

这些命令不直接显示在命令栏上，相反，在单击显示的三个点时，

它们显示为下拉框。

23.4.5 管理状态

与桌面应用程序不同，应用程序必须能随时暂停。在用户切换到另一个应用程序或桌面时，就会发生这种情况。所以它必须是一个由所有应用程序处理的很常见的场景。当应用程序暂停时，Windows将保存变量的值和数据结构，并在应用程序恢复执行时恢复它们。然而，应用程序可能已经暂停了较长时间，所以如果数据（如新闻摘要）随着时间的推移而变化，就应该在应用程序恢复执行时更新它。

暂停应用程序时，还应该考虑调用应用程序之间应保存的任何数据，如果应用程序随后由Windows或用户终止了，就没有机会这样做。

应用程序要暂停时，会发送Suspending事件，应该处理该事件。当应用程序恢复执行时，它将接收Resuming事件。处理这两个事件，保存应用程序的状态，就可以把应用程序返回到暂停之前的状态，用户不会注意到什么。

试一试：从暂停中恢复： **Ch23Ex06**

(1) 回到前面的示例。创建一个新类AppState:

```
using System.Collections.Generic;  
namespace BasicNavigation
```

```

{
    public static class AppState
    {
        private static Dictionary<string, bool> state = new Dictio
        public static bool GetState(string pageName) => state.Cont
state[pageName] : false;
        public static void SetState(string pageName, bool isBold)
        {
            if (state.ContainsKey(pageName))
                state[pageName] = isBold;
            else
                state.Add(pageName, isBold);
        }
        public static void Save()
        {
            var settings = Windows.Storage.ApplicationData.Current.R
            foreach (var key in state.Keys)
            {
                settings.Values[key] = state[key];
            }
        }
        public static void Load(string pageName)
        {
            if (!state.ContainsKey(pageName) &&
Windows.Storage.ApplicationData.Current.RoamingSettings.Values.Co
                state.Add(pageName,
(bool)Windows.Storage.ApplicationData.Current.RoamingSettings.Val

```

```

    }
}
}

```

(2) 打开app.xaml文件的代码隐藏文件，在底部定位OnSuspending方法。添加AppState.Save()，如下：

```

private void OnSuspending(object sender, SuspendingEventArgs
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background ac
    AppState.Save();
    deferral.Complete();
}

```

(3) 在OnLaunched方法的底部、Window.Current.Activate();的前面添加如下代码：

```

AppState.Load(typeof(BlankPage1).Name);
AppState.Load(typeof(BlankPage2).Name);
AppState.Load(typeof(BlankPage3).Name);

```

(4) 进入BlankPage1，在Page类上添加loaded事件，如下：

```

<Page
    x:Class="BasicNavigation.BlankPage1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/preser
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BasicNavigation"

```



```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d" Loaded="Page_Loaded">
```

(5) 实现事件：

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
    toggleButtonBold.IsChecked = AppState.GetState(GetType().Name,
    AppBarToggleButtonBold_Click(toggleButtonBold, new RoutedEventArgs())
}
```

(6) 修改开关按钮的单击事件处理程序，在按下按钮时保存页面的状态：

```
private void AppBarToggleButtonBold_Click(object sender, RoutedEventArgs e)
{
    AppState.SetState(GetType().Name, (bool)toggleButtonBold.IsChecked);
    ...
}
```

(7) 给BlankPage2和BlankPage3重复步骤4到6。

(8) 在app.xaml代码隐藏文件中，给OnSuspending方法设置一个断点。

(9) 运行该应用程序。

(10) 一旦运行应用程序，单击一两个页面上的Bold按钮。然后，应用程序仍在运行，返回Visual Studio。请注意，此时显示了Debug

Location工具栏，其中有一个Lifecycle Events下拉框。扩展它并单击Suspend。

(11) 一旦单步执行OnSuspended方法，应用程序就暂停。再次展开下拉框，单击Resume。

示例的说明

AppState类使用Windows.Storage.ApplicationData类来保存应用程序设置。这个类允许访问应用程序数据存储，轻松地设置一些简单的值。应该只在这个库中存储简单类型，如果需要保存应用程序中非常复杂的状态，应该考虑其他一些机制，如数据库或XML文件。

应用程序已经在app.xaml代码隐藏代码文件中处理Suspending事件，所以可以简单地使用它。如果必须为某些页面单独暂停事件，也应该在页面本身处理这个事件。

在OnSuspending事件中，保存了整个应用程序的状态，以便在应用程序重新启动时可以检索它。由于在应用程序从暂停中恢复时，任何页面都没有必须更新的数据，所以没有处理Resuming事件。

在OnLaunched方法加载应用程序时，恢复状态，OnLaunched方法也在app.xaml代码隐藏文件中。

23.5 Windows Store应用程序的常见元素

所有Windows Store应用程序都应该提供自己的磁贴和徽章。磁贴把应用程序显示在Windows的开始页面上，允许显示关于应用程序的信息。徽章允许Windows显示一个小图像，代表锁定屏幕上的应用程序。

磁贴很重要，因为用户往往很浮躁，倾向于根据应用程序如何显示自己来做出决策。同时，磁贴应该容易辨认，如果让用户搜索消失在一个磁贴中的另一个磁贴，在最终找到它之前，他们不太可能有好心情。

在Windows Store应用程序中有许多可能的磁贴大小，如果应用程序面向许多不同的显示屏尺寸，就应该按照所有建议的尺寸提供定制的磁贴，或者至少提供缩放得很好的磁贴。如果没有提供正确大小的磁贴，Windows会把所提供的磁贴缩放到正确的尺寸，但它通常看起来很糟糕。因此，对于专业的应用程序，确保使磁贴的每个尺寸都符合预期。

徽章比磁贴小（24×24像素），Windows显示应用程序时在锁定屏幕上使用。如果为应用程序设置了一个徽章图片，就必须启用Lock Screen通知。徽章也可以缩放，所以应提供所有适当的尺寸。

应用程序加载时，显示闪屏；因为闪屏应该只显示很短的时间，不应该太复杂，或给用户提供任何类型的信息，只需要清晰地表示应用程序正在启动。闪屏是620×300像素，但是可以使部分图像透明，让闪屏更小。另外，应提供按比例缩放的版本。

最后，应该提供一个50×50像素的“商店标志”，当然还应提供按比例缩放的本。

磁贴、徽章、标志嵌入在应用程序包清单中，在Visual Studio Manifest Package编辑器中可以轻松地编辑它。如果已经下载了本书的代码，就可以使用随代码一起提供的磁贴和徽章（在Assets文件夹中），否则可以在Paint或类似的应用程序中很快地创建图像。

试一试：添加磁贴和徽章

（1）使用Paint等图像编辑器创建如下尺寸的PNG图像：

- 620×300
- 310×150
- 310×310
- 150×150
- 71×71
- 50×50
- 44×44
- 24×24

给图片命名，使图片在不打开的情况也可以识别出来。

（2）打开前面示例中的项目。

（3）在Solution Explorer中双击Package.appxmanifest，打开包编辑器。

(4) 在Visual Assets标题下，找到左边的菜单，其中的选项可供改变磁贴、标志和闪屏。点击缩放100的图片按钮，浏览它们，添加图像。

(5) 在Solution Explorer中右击该项目，并选择Deploy。

示例的说明

进入“开始”菜单，找到应用程序。可能要单击“所有应用程序”或搜索名字。注意到小磁贴显示在列表中。如果右击它，并把应用程序固定到“开始”菜单中，就会使用一个更大的磁贴。可以右击磁贴，并选择Resize，改变其大小。

当应用程序运行时，闪屏会很快出现。

在菜单中右击应用程序，再次选择Uninstall，删除它。

23.6 Windows Store

创建应用程序之后，可能想将它分发给公众，方法是使用Windows Store。Microsoft竭尽全力，创建了一个安全的商店，让Windows用户从中下载应用程序，不必担心会下载恶意代码。不幸的是，这意味着把应用程序放在商店中必须经历漫长的过程。

23.6.1 打包应用程序

指定访问Pictures Library所需的Picture Viewer时，以及给应用程序添加磁贴时，都看到了package.appxmanifest文件的一些内容。准备打包应用程序，用于App Store时，必须回到这个文件，设置许多其他值。

打包应用程序之前，应该进入6个标签中，配置package.appxmanifest，考虑每一个选项：

- **Application:** 好好给应用程序命名！应用程序名和商店标志可能是潜在用户看到应用程序的第一个信息，所以给它指定一般的名字是无效的。试着选择一个有趣的名称，同时表示应用程序的作用。
- **Visual Assets:** 在最后一个示例中给应用程序添加了磁贴，应该确保在Visual Assets标签中，每一类别至少有一个图像。
- **Capabilities:** 在这个选项卡上，指定应用程序需要哪些功能。注意，如果应用程序需要的功能不合理，用户会把该应用程序视为可疑。例如，如果需要访问设备上的聊天信息，最好有一个充分的理由，否则很可能会被视为潜在地侵犯隐私。大多数应用程序都只需

要几个功能，但必须选择所有要使用的功能。如果没有准确地指定需要的功能，应用程序试图访问资源时，会收到一个拒绝访问异常。

- **Declarations:** 在这个选项卡上，可以把应用程序注册为服务提供者。例如，如果应用程序是一个搜索提供者，就可以将这个声明添加到应用程序中，并指定所需的属性。
- **Content URIs:** 如果应用程序导航到远程页面，它对系统的访问权限就是有限制的。可以使用Content URIs给Web页面提供定位设备和剪贴板的访问权限。
- **Packaging:** 这个选项卡可以设置包的属性，包括开发者/出版商的名称、应用程序的版本、用于签名包的证书。

23.6.2 创建包

一旦在appxmanifest中指定了所有需要的内容，就可以准备打包应用程序了。为此，可在Visual Studio中直接选择Store|Create App Packages。这将启动Create App Packages向导。在该向导的几步中，需要用store账户登录。如果没有store账户，就必须创建一个。必须有一个store账户，才能把应用程序发布到商店中，获得应用程序的报酬。

在向导中，会显示Select and Configure Packages页面。在这个页面中，一定要选择全部三个目标架构（x86、x64和ARM），应用程序才能部署到范围最广泛的设备上。

在最终页上，要选择如何确认应用程序可以提交给应用商店。启动Windows App Certification Kit，了解应用程序是否已准备好提交。如果检测到任何问题，就必须更正它们，再次执行Create App Packages向

导。如果应用程序通过检查，就可以上传包。

23.7 练习

(1) 扩展Ch23Ex06例子，给BlankPage1页面添加一个WebView控件，使用导航方法，显示所选的Web页面。给页面添加一个事件处理程序，在应用程序从暂停中恢复时，把webView导航到另一个网页。

(2) 如果希望应用程序用作录音机，就必须确保该应用程序可以访问设备上的麦克风。当应用程序试图使用设备上的麦克风时，如何确保它不会得到UnauthorizedAccessException？

(3) 运行在Windows Phone上的许多应用程序都使用一个称为Pivot的导航风格。也可以自己创建使用这种风格的通用应用程序。创建一个应用程序，使用Pivot控件显示三个视图，一个视图显示一个Web页面，另一个视图显示文本“Hello Pivot!”，第三个视图显示Wrox标志。该标志可以在http://media.wiley.com/assets/253/59/wrox_logo.gif上找到。

23.8 本章要点

主题	要点
Windows通用应用程序的XAML	Windows通用应用程序XAML和C#一起使用，创建Windows通用应用程序的GUI。它包括许多与WPF相同的控件，但是一些已经改变了，其他消失了，引入了新的控件
Visual State管理器	我们学习了如何使用Visual State管理器，只要改变控件的可视化状态，就可以改变控件和页面的外观。这样代码更少，但XAML略微复杂一些
应用程序状态	当用户切换到另一个应用或桌面时，Windows通用应用程序会暂停，所以一定要处理这种暂停，并在暂停时保存应用程序的状态
App store账户	这个账户用于把应用程序部署到Windows Store上
导航	在Windows通用应用程序中导航的方式几乎与Web应用程序相同，也使用方法调用在页面结构中来回移动

附录A 习题答案

第1、2章没有习题。

第3章

习题1

`super.smashing.great`

习题2

b)，因为它以数字开头；e)，因为它包含一个句点。

习题3

不，理论上没有限制包含在`string`变量中的字符串的长度。

习题4

这里，`*`和/以及`%`运算符的优先级最高，其次是`+`，最后是`+=`。本

习题中的优先级可以用括号来演示，如下所示：

```
resultVar += ((var1 * var2) + ((var3 % var4) / var5));
```

习题5

```
using static System.Console;  
using static System.Convert;  
static void Main(string[] args)  
{  
    int firstNumber, secondNumber, thirdNumber, fourthNumber;
```

```
    WriteLine("Give me a number:");
```

```
    firstNumber = ToInt32(ReadLine());
```

```
    WriteLine("Give me another number:");
```

```
    secondNumber = ToInt32(Console.ReadLine());
```

```
WriteLine("Give me another number:");
```

```
thirdNumber =.ToInt32(ReadLine());
```

```
WriteLine("Give me another number:");
```

```
fourthNumber =.ToInt32(ReadLine());
```

```
WriteLine($"The product of {firstNumber}, {secondNumber}, "
```

```
    $"{thirdNumber}, and {fourthNumber} is " +
```

```
        $"{firstNumber} * {secondNumber} * {thirdNumber} * {fourthN  
  
    }
```

注意这里使用Convert.ToInt32(), 本章没有介绍它们。

第4章

习题1

```
(var1 > 10) ^ (var2 > 10)
```

习题2

```
using static System.Console;  
using static System.Convert;  
static void Main(string[] args)  
{  
    bool numbersOK = false;  
    double var1, var2;  
    var1 = 0;  
    var2 = 0;  
    while (!numbersOK)  
    {
```

```

WriteLine("Give me a number:");
var1 = ToDouble(ReadLine());
WriteLine("Give me another number:");
var2 = ToDouble(ReadLine());
if ((var1 > 10) && (var2 > 10))
{
    numbersOK = true;
}
else
{
    if ((var1<= 10) && (var2<= 10))
    {
        numbersOK = true;
    }
    else
    {
        WriteLine("Only one number may be greater than 10.");
    }
}
}
WriteLine($"You entered {var1} and {var2}.");
}

```

注意使用另一个逻辑，执行效果更好，代码如下所示：

```

static void Main(string[] args)
{

```

```

bool numbersOK = false;
    double var1, var2;
    var1 = 0;
    var2 = 0;
    while (!numbersOK)
    {
        WriteLine("Give me a number:");
        var1 = ToDouble(ReadLine());
        WriteLine("Give me another number:");
        var2 = Convert.ToDouble(ReadLine());
        if ((var1 > 10) && (var2 > 10))
        {
            WriteLine("Only one number may be greater than 10.");
        }
        else
        {
            numbersOK = true;
        }
    }
    WriteLine($"You entered {var1} and {var2}.");
}

```

习题3

代码如下：

```
int i;
```



```
for (i = 1; i<= 10; i++)  
{  
    if ((i % 2) == 0)  
        continue;  
    WriteLine(i);  
}
```

使用赋值运算符=而不是布尔运算符==，这是一个十分常见的错误。

第5章

习题1

a)和c)的转换不能隐式进行。

习题2

```
enum color : short  
{  
    Red, Orange, Yellow, Green, Blue, Indigo, Violet, Black, W  
}
```

可以，byte类型可以包含0~255之间的数字。如果枚举项使用不同值，基于byte的枚举可以包含256项；如果给枚举项使用重复的值，就可以包含更多的项。

习题3

无法编译代码，原因如下：

- 遗漏了语句末尾的分号。
- 第二行尝试访问blab中不存在的第6个元素。
- 第二行尝试指定未包含在双引号中的字符串。

习题4

```
using static System.Console;
static void Main(string[] args)
{
    WriteLine("Enter a string:");

    string myString = ReadLine();

    string reversedString = "";

    for (int index = myString.Length - 1; index >= 0; index--)
```

```
{
```

```
    reversedString += myString[index];
```

```
}
```

```
    WriteLine($
```

```
"Reversed: {reversedString}
```

```
");
```

```
}
```

习题5

```
using static System.Console;  
static void Main(string[] args)  
{  
    WriteLine(  

```

```
"Enter a string:  

```

```
");  

```

```
    string myString = ReadLine();  

```

```
    myString = myString.Replace(  

```

```
"no  

```

```
",  

```

"yes

");

WriteLine(\$

"Replaced \

"no\

" with \

"yes\

": {myString}

");

}

习题6

```
using static System.Console;  
static void Main(string[] args)  
{  
    WriteLine("Enter a string:");
```

```
    string myString = ReadLine();
```

```
    myString = "\"" + myString.Replace(" ", "\" \") + "\"";
```

```
    WriteLine($"Added double quotes around words: {myString}")
```

```
}
```

或者使用String.Split():

```
using static System.Console;
static void Main(string[] args)
{
    WriteLine("Enter a string:");
    string myString = ReadLine();
    string[] myWords = myString.Split(' ');

    WriteLine("Adding double quotes around words:");

    foreach (string myWord in myWords)

    {

        Write($"\"{myWord}\" ");
    }
}
```

```
}
```

```
}
```

第6章

习题1

第一个函数的返回类型是bool，但不返回一个bool值。

第二个函数有一个params实参，但这个实参不在实参列表的末尾处。

习题2

```
using static System.Console;  
static void Main(string[] args)  
{
```



```
if (args.Length != 2)
```

```
{
```

```
    WriteLine("Two arguments required.");
```

```
    return;
```

```
}
```

```
string param1 = args[0];
```

```
int param2 =.ToInt32(args[1]);
```

```

        WriteLine($"String parameter: {param1}",);

        WriteLine($"Integer parameter: {param2}",);

    }

```

注意这个答案包含的代码检查是否提供了两个实参，本题没有这个要求，但在本题中进行检查是很合理的。

习题3

```

class Program
{
    using static System.Console;
    delegate string ReadLineDelegate();

    static void Main(string[] args)
    {
        ReadLineDelegate readLine = new ReadLineDelegate(ReadLine

```

```
WriteLine("Type a string:");
```

```
string userInput = readLine();
```

```
WriteLine($"You typed: {userInput}");
```

```
}
```

```
}
```

习题4

```
struct order
```

```
{
```

```
    public string itemName;
```

```
    public int unitCount;
```

```
    public double unitCost;
```

```
    public double TotalCost() => unitCount * unitCost;
```

```
}
```

习题5

```
struct order
{
    public string itemName;
    public int unitCount;
    public double unitCost;
    public double TotalCost() => unitCount * unitCost;
    public string Info() => "Order information: " + unitCount.T

    " " + itemName + " items at $" + unitCost.ToString() +

    " each, total cost $" + TotalCost().ToString();

}
```

第7章

习题1

这个语句仅对要用于所有版本的信息有效。而且，我们常常希望仅在使用调试版本时输出调试信息。此时应首选`Debug.WriteLine()`版本。

使用`Debug.WriteLine()`版本还有一个优点：该版本不会编译到发布版本中，从而使最终代码文件变得更小。

习题2

```
static void Main(string[] args)
{
    for (int i = 1; i<10000; i++)

{

    WriteLine($"Loop cycle {i}");

    if (i == 5000)
```

```
{  
  
    WriteLine(args[999]);  
  
}  
  
}  
  
}
```

在VS中，可以把断点放在下面的代码行上：

```
WriteLine("Loop cycle {0}", i);
```

应修改断点的属性，把执行次数的条件设置为“执行次数等于5000时中断”。

习题3

错误，finally块始终会执行，它可能在处理catch块之后执行。

习题4

```
static void Main(string[] args)
{
    Orientation myDirection;

    for (byte myByte = 2; myByte<10; myByte++)

    {

        try

        {
```

```
myDirection = checked((Orientation)myByte);
```

```
if ((myDirection<Orientation.North)
```

```
||
```

```
(myDirection > Orientation.West))
```

```
{
```

```
throw new ArgumentOutOfRangeException("myByte", myByt
```

```
"Value must be between 1 and 4");
```

```
}
```



```
}
```

```
catch (ArgumentOutOfRangeException e)
```

```
{
```

```
// If this section is reached then myByte<1 or myByte >
```

```
WriteLine(e.Message);
```

```
WriteLine("Assigning default value, Orientation.North.")
```

```
        myDirection = Orientation.North;

    }

    WriteLine($"myDirection = {myDirection}");

}

}
```

注意这是一个小问题。因为枚举基于byte类型，所以可给其赋予任意byte值，即使在枚举中没有为该值指定名称也同样如此。在上面的代码中，如有必要，可以生成自己的异常。

第8章

习题1

b、d和e。public、private和protected是实际的可访问级别。

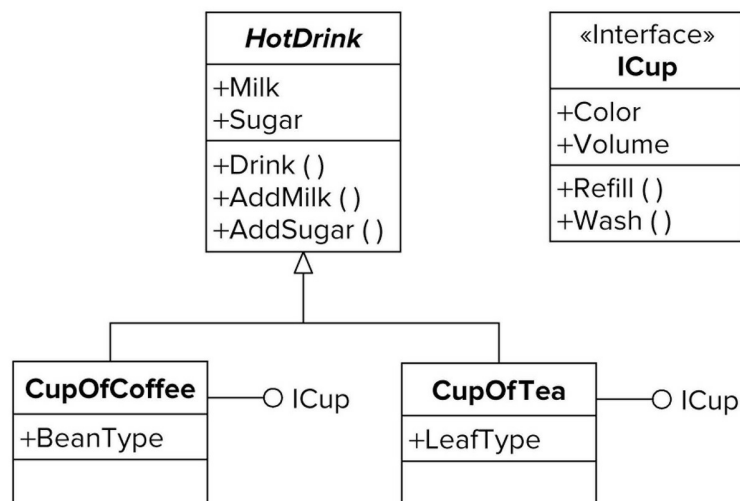
习题2

错误，永远都不应手工调用对象的析构函数，.NET运行库环境会在垃圾回收过程中自动完成该任务。

习题3

不，可以在没有任何类实例的情况下调用静态方法。

习题4



图A-1

习题5

```

static void ManipulateDrink(HotDrink drink)
{
    drink.AddMilk();
    drink.Drink();
    ICup cupInterface = (ICup)drink;
    cupInterface.Wash();
}

```

注意显式转换为ICup的代码行。这是必需的，因为HotDrink不支持ICup接口，但我们知道传送给这个函数的两个cup对象支持ICup接口。这很危险，因为也可以给这个函数传送其他类，但这些类也可能派生于HotDrink，而HotDrink却不支持ICup接口。为更正这个问题，应检查该接口是否得到支持：

```

static void ManipulateDrink(HotDrink drink)
{
    drink.AddMilk();
    drink.Drink();
    if (drink is ICup)

    {

        ICup cupInterface = drink as ICup;
    }
}

```

```
        cupInterface.Wash();  
  
    }  
  
}
```

这里使用的is和as操作符在第11章介绍。

第9章

习题1

myDerivedClass派生于MyClass，但是MyClass是密封的，不能从MyClass中派生其他类。

习题2

要定义不能创建的类，可以将其定义为静态类，或者将其所有构造函数定义为私有。

习题3

不能创建的类可通过它们拥有的静态成员来使用。实际上，甚至可以通过这些成员获取这些类的实例，如下所示：

```
class CreateMe
{
    private CreateMe()
    {
    }
    static public CreateMe GetCreateMe()
    {
        return new CreateMe();
    }
}
```

这里，公共构造函数可以访问私有构造函数，因为它在同一个类的定义中。

习题4

为简单起见，下面的类定义显示为一个代码文件的一部分，而没有给每个类定义列出单独的代码文件：

```
namespace Vehicles
{
    public abstract class Vehicle
```

```
{
```

```
}
```

```
public abstract class Car : Vehicle
```

```
{
```

```
}
```

```
public abstract class Train : Vehicle
```

```
{
```

```
}
```

```
public interface IPassengerCarrier
```

```
{
```

```
}
```

```
public interface IHeavyLoadCarrier
```

```
{
```



```
}
```

```
public class SUV : Car, IPassengerCarrier
```

```
{
```

```
}
```

```
public class Pickup : Car, IPassengerCarrier, IHeavyLoadCar
```

```
{
```

```
}
```

```
public class Compact : Car, IPassengerCarrier
```

```
{
```

```
}
```

```
public class PassengerTrain : Train, IPassengerCarrier
```

```
{
```

```
}
```

```
public class FreightTrain : Train, IHeavyLoadCarrier
```

```
{
```

```
}
```

```
public class T424DoubleBogey : Train, IHeavyLoadCarrier
```

```
{
```

```
}
```

```
}
```

习题5

```
using System;  
using static System.Console;  
using Vehicles;
```

```
namespace Traffic  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            AddPassenger(new Compact());  
  
            AddPassenger(new SUV());  
  
            AddPassenger(new Pickup());  
  
            AddPassenger(new PassengerTrain());
```

```
ReadKey();
```

```
}
```

```
static void AddPassenger(IPassengerCarrier Vehicle)
```

```
{
```

```
WriteLine(Vehicle.ToString());
```

```
}
```

```
}
```

```
}
```

第10章

习题1

```
class MyClass
{
    protected string myString;
    public string ContainedString
    {
        set
        {
            myString = value;
        }
    }
    public virtual string GetString() => myString;
}
```

习题2

```
class MyDerivedClass : MyClass
{
    public override string GetString() => base.GetString() +
        " (output from derived class)";
}
```

习题3

如果方法具有返回类型，就可以将其用作表达式的一部分：

```
x = Manipulate(y, z);
```

如果没有给部分方法提供实现代码，编译器就会在使用该部分方法的所有地方删除该方法。在上面的代码中，这会使x的结果变得模糊，因为Manipulate()方法没有替代方法。如果没有这个方法，可能只需要忽略整行代码，但编译器无法确定我们是否的确希望忽略它。

没有返回类型的方法不能作为表达式的一部分来调用，所以编译器可以安全地删除对部分方法调用的所有引用。

同样，也禁止使用out参数，因为在方法调用之前，用作out参数的变量必须是未定义的，而应在方法调用之后定义。删除方法调用会违反这个规则。

习题4

```
class MyCopyableClass
{
    protected int myInt;
    public int ContainedInt
    {
        get
        {
            return myInt;
        }
        set
        {
```

```

        myInt = value;
    }
}
public MyCopyableClass GetCopy() => (MyCopyableClass)Member
}

```

客户端代码:

```

class Program
{
    using static System.Console;
    static void Main(string[] args)
    {
        MyCopyableClass obj1 = new MyCopyableClass();

        obj1.ContainedInt = 5;

        MyCopyableClass obj2 = obj1.GetCopy();

        obj1.ContainedInt = 9;
    }
}

```



```
        WriteLine(obj2.ContainedInt);  
  
    }  
}
```

这些代码显示5，说明所复制对象有自己专用的myInt字段。

习题5

```
using System;  
using static System.Console;  
using Ch10CardLib;  
  
namespace Exercise_Answers  
{  
    class Class1  
    {  
        static void Main(string[] args)  
        {  
            while(true)
```

```
{
```

```
    Deck playDeck = new Deck();
```

```
    playDeck.Shuffle();
```

```
    bool isFlush = false;
```

```
    int flushHandIndex = 0;
```

```
    for (int hand = 0; hand<10; hand++)
```

```
    {
```

```
isFlush = true;
```

```
Suit flushSuit = playDeck.GetCard(hand * 5).suit;
```

```
for (int card = 1; card<5; card++)
```

```
{
```

```
    if (playDeck.GetCard(hand * 5 + card).suit != flushSuit)
```

```
    {
```

```
isFlush = false;
```

```
}
```

```
}
```

```
if (isFlush)
```

```
{
```

```
flushHandIndex = hand * 5;
```

```
break;
```

```
}
```

```
}
```

```
if (isFlush)
```

```
{
```

```
WriteLine("Flush!");
```

```
for (int card = 0; card<5; card++)
```

```
{
```

```
WriteLine(playDeck.GetCard(flushHandIndex + card)),
```

```
}
```

```
}
```

```
else
```

```
{
```

```
WriteLine("No flush.");
```

```
}
```

```
        ReadLine());  
  
    }  
  
}  
  
}  
}
```

这些代码会循环下去，因为同花色是不常见的。可能需要按几次回车键，才能在洗好的扑克牌中找到一个同花色。为了验证一切如期执行，可以尝试将洗牌的代码行注释掉。

第11章

习题1

```
using System;  
using System.Collections;  
namespace Exercise_Answers
```

```

{
    public class People : DictionaryBase
    {
        public void Add(Person newPerson) =>
            Dictionary.Add(newPerson.Name, newPerson);
        public void Remove(string name) => Dictionary.Remove(name);
        public Person this[string name]
        {
            get
            {
                return (Person)Dictionary[name];
            }
            set
            {
                Dictionary[name] = value;
            }
        }
    }
}

```

习题2

```

public class Person
{
    private string name;
    private int age;
}

```



```
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
public int Age
{
    get
    {
        return age;
    }
    set
    {
        age = value;
    }
}
public static bool operator >(Person p1, Person p2) =>
    p1.Age > p2.Age;
public static bool operator <(Person p1, Person p2) =>
    p1.Age < p2.Age;
public static bool operator >=(Person p1, Person p2) =>
```

```

        !(p1<p2);
    public static bool operator<=(Person p1, Person p2) =>
        !(p1 > p2);
}

```

习题3

```

public Person[] GetOldest()
{
    Person oldestPerson = null;
    People oldestPeople = new People();
    Person currentPerson;
    foreach (DictionaryEntry p in Dictionary)
    {
        currentPerson = p.Value as Person;
        if (oldestPerson == null)
        {
            oldestPerson = currentPerson;
            oldestPeople.Add(oldestPerson);
        }
        else
        {
            if (currentPerson > oldestPerson)
            {
                oldestPeople.Clear();
                oldestPeople.Add(currentPerson);
            }
        }
    }
}

```

```

        oldestPerson = currentPerson;
    }
    else
    {
        if (currentPerson >= oldestPerson)
        {
            oldestPeople.Add(currentPerson);
        }
    }
}
}
Person[] oldestPeopleArray = new Person[oldestPeople.Count];
int copyIndex = 0;
foreach (DictionaryEntry p in oldestPeople)
{
    oldestPeopleArray[copyIndex] = p.Value as Person;
    copyIndex++;
}
return oldestPeopleArray;
}

```

这个函数比较复杂，因为没有为**Person**定义==运算符，但仍可以构建逻辑。另外，返回**People**实例更加简单，因为在处理过程中比较容易操作这个类。作为一个折中方法，在整个函数中都使用了**People**实例，再在最后转换为一个**Person**实例数组。

习题4

```

public class People : DictionaryBase, ICloneable
{
    public object Clone()
    {
        People clonedPeople = new People();
        Person currentPerson, newPerson;
        foreach (DictionaryEntry p in Dictionary)
        {
            currentPerson = p.Value as Person;
            newPerson = new Person();
            newPerson.Name = currentPerson.Name;
            newPerson.Age = currentPerson.Age;
            clonedPeople.Add(newPerson);
        }
        return clonedPeople;
    }
    ...
}

```

在Person类上实现ICloneable接口，可以简化这段代码。

习题5

```

public IEnumerable Ages
{
    get
    {

```

```
        foreach (object person in Dictionary.Values)
            yield return (person as Person).Age;
    }
}
```

第12章

习题1

a、b和e是

c和d否，但它们可以使用由包含它们的类提供的泛型类型参数。

f否。

习题2

```
public static double? operator *(Vector op1, Vector op2)
{
    try
    {
        double angleDiff = (double)(op2.ThetaRadians.Value -
            op1.ThetaRadians.Value);
        return op1.R.Value * op2.R.Value * Math.Cos(angleDiff);
    }
    catch
```

```
{
    return null;
}
}
```

习题3

不在T上强制new()约束，就不能实例化T。在T上强制new()约束可以确保有一个公共的默认构造函数是可用的。

```
public class Instantiator<T>
    where T : new()
{
    public T instance;
    public Instantiator()
    {
        instance = new T();
    }
}
```

习题4

同一个泛型类型参数T既用于泛型类，又用于泛型方法。需要重命名其中的一个或两个。例如：

```
public class StringGetter<U>
{

```

```
        public string GetString<T>(T item) => item.ToString();  
    }  
}
```

习题5

一种方式如下：

```
public class ShortList<T> : IList<T>  
{  
    protected IList<T> innerCollection;  
    protected int maxSize = 10;  
    public ShortList()  
        : this(10)  
    {  
    }  
    public ShortList(int size)  
    {  
        maxSize = size;  
        innerCollection = new List<T>();  
    }  
    public ShortList(IEnumerable<T> list)  
        : this(10, list)  
    {  
    }  
    public ShortList(int size, IEnumerable<T> list)  
    {  
        maxSize = size;
```

```

        innerCollection = new List<T>(list);
        if (Count > maxSize)
        {
            ThrowTooManyItemsException();
        }
    }

    protected void ThrowTooManyItemsException()
    {
        throw new IndexOutOfRangeException(
            "Unable to add any more items, maximum size is " + max
            + " items.");
    }

    #region IList<T> Members

    public int IndexOf(T item) => innerCollection.IndexOf(item)
    public void Insert(int index, T item)
    {
        if (Count < maxSize)
        {
            innerCollection.Insert(index, item);
        }
        else
        {
            ThrowTooManyItemsException();
        }
    }

    public void RemoveAt(int index)
    {

```



```

        innerCollection.RemoveAt(index);
    }
    public T this[int index]
    {
        get
        {
            return innerCollection[index];
        }
        set
        {
            innerCollection[index] = value;
        }
    }
}
#endregion

#region ICollection<T> Members
public void Add(T item)
{
    if (Count<maxSize)
    {
        innerCollection.Add(item);
    }
    else
    {
        ThrowTooManyItemsException();
    }
}
}
public void Clear()

```

```

{
    innerCollection.Clear();
}
public bool Contains(T item) => innerCollection.Contains(item)
public void CopyTo(T[] array, int arrayIndex)
{
    innerCollection.CopyTo(array, arrayIndex);
}
public int Count
{
    get
    {
        return innerCollection.Count;
    }
}
public bool IsReadOnly
{
    get
    {
        return innerCollection.IsReadOnly;
    }
}
public bool Remove(T item) => innerCollection.Remove(item);
#endregion
#region IEnumerable<T> Members
public IEnumerator<T> GetEnumerator() =>
    innerCollection.GetEnumerator();

```

```

#endregion

#region IEnumerable Members

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

#endregion
}

```

习题6

不。类型参数T定义为协变。但协变参数类型只能用作方法的返回值，不能用作方法实参。否则就会得到如下编译错误（假定使用名称空间VarianceDemo）：

```

Invalid variance: The type parameter 'T' must be contravariant
'VarianceDemo.IMethaneProducer<T>.BelchAt(T)'. 'T' is covariant

```

第13章

习题1

```

using static System.Console;

public void ProcessEvent(object source, EventArgs e)
{
    if (e is MessageArrivedEventArgs)
    {
        WriteLine("Connection.MessageArrived event received.");
        WriteLine($"Message: {(e as MessageArrivedEventArgs).Message}");
    }
}

```

```

    }
    if (e is ElapsedEventArgs)
    {
        WriteLine("Timer.Elapsed event received.");
        WriteLine($"SignalTime: {(e as ElapsedEventArgs ).SignalT:
    }
}

```

习题2

修改Player.cs，如下所示（修改了一个方法，添加了两个新方法——代码中的注释说明了这些变化）：

```

public bool HasWon()
{
    // get temporary copy of hand, which may get modified.
    Cards tempHand = (Cards)PlayHand.Clone();
    // find three and four of a kind sets
    bool fourOfAKind = false;
    bool threeOfAKind = false;
    int fourRank = -1;
    int threeRank = -1;
    int cardsOfRank;
    for (int matchRank = 0; matchRank<13; matchRank++)
    {
        cardsOfRank = 0;
        foreach (Card c in tempHand)

```

```

{
    if (c.rank == (Rank)matchRank)
    {
        cardsOfRank++;
    }
}
if (cardsOfRank == 4)
{
    // mark set of four
    fourRank = matchRank;
    fourOfAKind    if (cardsOfRank == 3)
{
    // two threes means no win possible
    // (threeOfAKind will be true only if this code
    // has already executed)
    if (threeOfAKind == true)
    {
        return false;
    }
    // mark set of three
    threeRank = matchRank;
    threeOfAKind = true;
}
}
// check simple win condition
if (threeOfAKind && fourOfAKind)
{

```

```

        return true;
    }
    // simplify hand if three or four of a kind is found,
    // by removing used cards
    if (fourOfAKind || threeOfAKind)
    {
        for (int cardIndex = tempHand.Count - 1; cardIndex >= 0; cardIndex--)
        {
            if ((tempHand[cardIndex].rank == (Rank)fourRank)
                || (tempHand[cardIndex].rank == (Rank)threeRank))
            {
                tempHand.RemoveAt(cardIndex);
            }
        }
    }
    // at this point the method may have returned, because:
    // - a set of four and a set of three has been found, winning
    // - two sets of three have been found, losing.
    // if the method hasn't returned then:
    // - no sets have been found, and tempHand contains 7 cards
    // - a set of three has been found, and tempHand contains 4
    // - a set of four has been found, and tempHand contains 3
    // find run of four sets, start by looking for cards of same suit
    // in the same way as before
    bool fourOfASuit = false;
    bool threeOfASuit = false;
    int fourSuit = -1;

```

```

int threeSuit = -1;
int cardsOfSuit;
for (int matchSuit = 0; matchSuit<4; matchSuit++)
{
    cardsOfSuit = 0;
    foreach (Card c in tempHand)
    {
        if (c.suit == (Suit)matchSuit)
        {
            cardsOfSuit++;
        }
    }
    if (cardsOfSuit == 7)
    {
        // if all cards are the same suit then two runs
        // are possible, but not definite.
        threeOfASuit = true;
        threeSuit = matchSuit;
        fourOfASuit = true;
        fourSuit = matchSuit;
    }
    if (cardsOfSuit == 4)
    {
        // mark four card suit.
        fourOfASuit = true;
        fourSuit = matchSuit;
    }
}

```

```

    if (cardsOfSuit == 3)
    {
        // mark three card suit.
        threeOfASuit = true;
        threeSuit = matchSuit;
    }
}
if (!(threeOfASuit || fourOfASuit))
{
    // need at least one run possibility to continue.
    return false;
}
if (tempHand.Count == 7)
{
    if (!(threeOfASuit && fourOfASuit))
    {
        // need a three and a four card suit.
        return false;
    }
    // create two temporary sets for checking.
    Cards set1 = new Cards();
    Cards set2 = new Cards();
    // if all 7 cards are the same suit...
    if (threeSuit == fourSuit)
    {
        // get min and max cards
        int maxVal, minVal;

```



```

GetLimits(tempHand, out maxVal, out minVal);
for (int cardIndex = tempHand.Count - 1; cardIndex >= 0;
{
    if (((int)tempHand[cardIndex].rank < (minVal + 3))
        || ((int)tempHand[cardIndex].rank > (maxVal - 3)))
    {
        // remove all cards in a three card set that
        // starts at minVal or ends at maxVal.
        tempHand.RemoveAt(cardIndex);
    }
}
if (tempHand.Count != 1)
{
    // if more than one card is left then there aren't two
    return false;
}
if ((tempHand[0].rank == (Rank)(minVal + 3))
    || (tempHand[0].rank == (Rank)(maxVal - 3)))
{
    // if spare card can make one of the three card sets i
    // four card set then there are two sets.
    return true;
}
else
{
    // if spare card doesn't fit then there are two sets o
    // cards but no set of four cards.

```

```

        return false;
    }
}
// if three card and four card suits are different...
foreach (Card card in tempHand)
{
    // split cards into sets.
    if (card.suit == (Suit)threeSuit)
    {
        set1.Add(card);
    }
    else
    {
        set2.Add(card);
    }
}
// check if sets are sequential.
if (isSequential(set1) && isSequential(set2))
{
    return true;
}
else
{
    return false;
}
}
// if four cards remain (three of a kind found)

```

```
if (tempHand.Count == 4)
{
    // if four cards remain then they must be the same suit.
    if (!fourOfASuit)
    {
        return false;
    }
    // won if cards are sequential.
    if (isSequential(tempHand))
    {
        return true;
    }
}
// if three cards remain (four of a kind found)
if (tempHand.Count == 3)
{
    // if three cards remain then they must be the same suit.
    if (!threeOfASuit)
    {
        return false;
    }
    // won if cards are sequential.
    if (isSequential(tempHand))
    {
        return true;
    }
}
```

```

    // return false if two valid sets don't exist.
    return false;
}
// utility method to get max and min ranks of cards
// (same suit assumed)
private void GetLimits(Cards cards, out int maxVal, out int minVal)
{
    maxVal = 0;
    minVal = 14;
    foreach (Card card in cards)
    {
        if ((int)card.rank > maxVal)
        {
            maxVal = (int)card.rank;
        }
        if ((int)card.rank < minVal)
        {
            minVal = (int)card.rank;
        }
    }
}
// utility method to see if cards are in a run
// (same suit assumed)
private bool isSequential(Cards cards)
{
    int maxVal, minVal;
    GetLimits(cards, out maxVal, out minVal);

```

```
if ((maxVal - minVal) == (cards.Count - 1))
{
    return true;
}
else
{
    return false;
}
}
```

习题3

为结合使用对象初始化和类，必须包含一个默认的空参构造函数。可以给这个类添

```
Giraffe myPetGiraffe = new Giraffe
{
    NeckLength = "3.14",
    Name = "Gerald"
};
```

习题4

错误。在使用**var**关键字来声明变量时，该变量仍是强类型化的，编译器会确定变量

习题5

可以使用已实现的**Equals()**方法。注意不能使用**==**运算符来执行这个操作，因为==

习题6

扩展方法必须是静态的：

```
public static string ToAcronym(this string inputString)
```

习题7

必须在静态类中包含扩展方法，可以在包含客户代码的名称空间中访问它。为此，可

习题8

一种方式如下：

```
public static string ToAcronym(this string inputString) =>
    inputString.Trim().Split(' ').Aggregate<string, string>("",
        (a, b) => a + (b.Length > 0 ?
            b.ToUpper()[0].ToString() : ""));
```

其中使用了三元运算符以防多个空格引发错误。还要注意需要带两个泛型类型参数的

第14章

习题1

将TextBlock控件放到一个ScrollView面板中。把VerticalScrollBarVisibi

```
<Window x:Class="Answers.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="14.1 Solution" Height="350" Width="525">
    <Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="75"/>
        <RowDefinition />
    </Grid.RowDefinitions>
    <Label Content="Enter text" HorizontalAlignment="Left" Mar
VerticalAlignment="Top"/>
        <TextBox HorizontalAlignment="Left" Margin="76,12,0,0" Te
VerticalAlignment="Top" Height="53" Width="423" AcceptsReturn=
Name="textTextBox">
        </TextBox>
        <ScrollView HorizontalAlignment="Left" Height="217" Marq
Grid.Row="1" VerticalAlignment="Top" Width="489"
```



```

VerticalScrollBarVisibility="Auto">
    <TextBlock TextWrapping="Wrap" Text="{Binding ElementName=
Path=Text}"/>
</ScrollViewer>
</Grid>
</Window>

```

习题2

把一个Slider和一个ProgressBar控件拖动到视图中后，将Slider控件的最小值

```

<Window x:Class="Answers. Ch14Solution2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="14.2 Solution" Height="300" Width="300">
    <Grid>
        <Slider HorizontalAlignment="Left" Margin="10,10,0,0" Veri
Width="264" Minimum="1" Maximum="100" Name="valueSlider"/>
        <ProgressBar HorizontalAlignment="Left" Height="24" Margi
VerticalAlignment="Top" Width="264"
Minimum="{Binding ElementName=valueSlider, Path=Minimum}"
Maximum="{Binding ElementName=valueSlider, Path=Maximum}"

```

```

Value="{Binding ElementName=valueSlider, Path=Value}"/>
    </Grid>
</Window>

```

习题3

可使用RenderTransform。在设计视图中，可将光标移到控件边缘处，看到一个四

```

<Window x:Class="Answers. Ch14Solution3"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="14.3 Solution" Height="300" Width="300">
    <Grid>
        <Slider HorizontalAlignment="Left" Margin="10,10,0,0" Vertic
Width="264" Minimum="1" Maximum="100" Name="valueSlider"/>
        <ProgressBar HorizontalAlignment="Left" Height="24" Margin="
VerticalAlignment="Top" Width="311"
Minimum="{Binding ElementName=valueSlider, Path=Minimum}" Maxi
ElementName=valueSlider, Path=Maximum}"
Value="{Binding ElementName=valueSlider, Path=Value}"
RenderTransformOrigin="0.5,0.5">
            <ProgressBar.RenderTransform>

```

```
<TransformGroup>
    <ScaleTransform/>
    <SkewTransform/>
    <RotateTransform Angle="-36.973"/>
    <TranslateTransform/>
</TransformGroup>
</ProgressBar.RenderTransform>
</ProgressBar>
</Grid>
</Window>
```

习题4

PersistentSlider类必须实现INotifyPropertyChanged接口。

创建一个字段，用于保存3个属性的值。

在属性的每个setter中，调用PropertyChanged事件的任意订阅者。为达到此目的

```
using System.ComponentModel;
```

```

namespace Answers
{
    public class PersistentSlider : INotifyPropertyChanged
    {
        private int _minValue;
        private int _maxValue;
        private int _currentValue;
        public int MinValue
        {
            get { return _minValue; }
            set { _minValue = value; OnPropertyChanged(nameof(MinVal
        }
        public int MaxValue
        {
            get { return _maxValue; }
            set { _maxValue = value; OnPropertyChanged(nameof(MaxVal
        }
        public int CurrentValue
        {
            get { return _currentValue; }
            set { _currentValue = value; OnPropertyChanged(nameof(Cu
        }
        public event PropertyChangedEventHandler PropertyChanged;
        protected void OnPropertyChanged(string propertyName) => {
            Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}

```

(1) 在代码隐藏文件中，添加如下字段：

```
private PersistentSlider _sliderData = new PersistentSlider  
MaxValue = 200, CurrentValue = 100 };
```

(2) 在构造函数中，将当前实例的DataContext属性设置为刚才创建的字段：

```
this.DataContext = _sliderData;  
InitializeComponent();
```

(3) 在XAML中，将Slider控件修改为使用数据上下文。只需要设置Path：

```
<Window x:Class="Answers. Ch14Solution4"  
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Title="14.4 Solution" Height="300" Width="300">  
    <Grid>  
        <Slider HorizontalAlignment="Left" Margin="10,10,0,0" Ver  
Width="264" Minimum="{Binding Path=MinValue}"  
Maximum="{Binding Path=MaxValue}" Value="{Binding Path=Current
```

```

Name="valueSlider"/>
    <ProgressBar HorizontalAlignment="Left" Height="24" Margin=
VerticalAlignment="Top" Width="311"
Minimum="{Binding ElementName=valueSlider, Path=Minimum}"
Maximum="{Binding ElementName=valueSlider, Path=Maximum}"
Value="{Binding ElementName=valueSlider, Path=Value}"
RenderTransformOrigin="0.5,0.5">
    <ProgressBar.RenderTransform>
        <TransformGroup>
            <ScaleTransform/>
            <SkewTransform/>
            <RotateTransform Angle="-36.973"/>
            <TranslateTransform/>
        </TransformGroup>
    </ProgressBar.RenderTransform>
</ProgressBar>
</Grid>
</Window>

```

第15章

习题1

(1) 创建一个新类，其名称为ComputerSkillValueConverter，如下所示：

```
using Ch13CardLib;
using System;
using System.Windows.Data;
namespace KarliCards_Gui
{
    [ValueConversion(typeof(ComputerSkillLevel), typeof(bool))]
    public class ComputerSkillValueConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
            System.Globalization.CultureInfo culture)
        {
            string helper = parameter as string;
            if (string.IsNullOrEmpty(helper))
                return false;

            ComputerSkillLevel skillLevel = (ComputerSkillLevel)value;
            return (skillLevel.ToString() == helper);
        }

        public object ConvertBack(object value, Type targetType, object parameter,
            System.Globalization.CultureInfo culture)
        {
            string parameterString = parameter as string;
            if (parameterString == null)
```

```

        return ComputerSkillLevel.Dumb;
    }
    return Enum.Parse<ComputerSkillLevel>(targetType, parameterString);
}
}
}

```

(2) 在Options.xaml中添加一个静态资源声明:

```

<Window.Resources>
    <src:ComputerSkillValueConverter x:Key="skillConverter" />
</Window.Resources>

```

(3) 修改单选按钮，如下所示:

```

<RadioButton Content="Dumb" HorizontalAlignment="Left"
Margin="37,41,0,0" VerticalAlignment="Top" Name="dumbAIRadioButton"
IsChecked="{Binding ComputerSkill, Converter={StaticResource skillConverter}, ConverterParameter=Dumb}" />
<RadioButton Content="Good" HorizontalAlignment="Left"
Margin="37,62,0,0" VerticalAlignment="Top" Name="goodAIRadioButton"
IsChecked="{Binding ComputerSkill, Converter={StaticResource skillConverter}, ConverterParameter=Good}" />
<RadioButton Content="Cheats" HorizontalAlignment="Left"

```



```
Margin="37,83,0,0" VerticalAlignment="Top" Name="cheatingAIRac  
IsChecked="{Binding ComputerSkill, Converter={StaticResource s  
ConverterParameter=Cheats}}" />
```

(4) 删除代码隐藏文件中的事件。

习题2

(1) 在Options.xaml对话框中添加一个新复选框:

```
<CheckBox Content="Plays with open cards" HorizontalAlignment=  
Margin="10,100, 0,0" VerticalAlignment="Top"  
IsChecked="{Binding ComputerPlaysWithOpenHand}" />
```

(2) 在GameOptions.cs类中添加一个新属性:

```
private bool _computerPlaysWithOpenHand;  
public bool ComputerPlaysWithOpenHand  
{
```

```

    get { return _computerPlaysWithOpenHand; }
    set
    {
        _computerPlaysWithOpenHand = value;
        OnPropertyChanged(nameof(ComputerPlaysWithOpenHand));
    }
}

```

(3) 在CardsInHandControl中添加一个新的依赖属性:

```

public bool ComputerPlaysWithOpenHand
{
    get { return (bool)GetValue(ComputerPlaysWithOpenHandProperty); }
    set { SetValue(ComputerPlaysWithOpenHandProperty, value); }
}

public static readonly DependencyProperty ComputerPlaysWithOpenHandProperty =
    DependencyProperty.Register("ComputerPlaysWithOpenHand",
        typeof(bool), typeof(CardsInHandControl), new PropertyMetadata(false));

```

(4) 在CardsInHandControl类的DrawCards方法中, 修改对isFaceUP所做的判断:

```

if (Owner is ComputerPlayer)
    isFaceup = (Owner.State == CardLib.PlayerState.Loser |

```

```
Owner.State == CardLib.PlayerState.Winner || ComputerPlaysWith
```

(5) 给GameViewModel添加一个新属性:

```
public bool ComputerPlaysWithOpenHand
{
    get { return _gameOptions.ComputerPlaysWithOpenHand; }
}
```

(6) 将新属性绑定到所有4个玩家游戏客户端的CardsInHandControls:

```
ComputerPlaysWithOpenHand="{Binding GameOptions.ComputerPlaysw
```

习题3

(1) 在GameViewModel中添加一个新属性, 如下所示:

```
private string _currentStatusText = "Game is not started",
```

```

public string CurrentStatusText
{
    get { return _currentStatusText; }
    set
    {
        _currentStatusText = value;
        OnPropertyChanged(nameof(CurrentStatusText));
    }
}

```

(2) 修改CurrentPlayer属性，如下所示：

```

public Player CurrentPlayer
{
    get { return _currentPlayer; }
    set
    {
        _currentPlayer = value;
        OnPropertyChanged("CurrentPlayer");
        if (!Players.Any(x => x.State == PlayerState.Winner))
        {
            Players.ForEach(x => x.State = (x == value ? PlayerState.Winner : PlayerState.Inactive));
            CurrentStatusText = $"Player {CurrentPlayer.PlayerName}";
        }
    }
}

```

```

        else
        {
            var winner = Players.Where(x => x.HasWon).FirstOrDefault();
            if (winner != null)
                CurrentStatusText = $"Player {winner.PlayerName} has won!";
        }
    }
}

```

(3) 在StartNewGame方法的末尾处添加下面的代码:

```

CurrentStatusText = string.Format("New game stated. Player {0} has won!",
CurrentPlayer.PlayerName);

```

(4) 在游戏客户端的XAML中添加一个状态栏, 将其绑定到新属性:

```

<StatusBar Grid.Row="3" HorizontalAlignment="Center" Margin="0, 10, 0, 10"
VerticalAlignment="Center" Background="Green" Foreground="White">
    <StatusBarItem VerticalAlignment="Center">
        <TextBlock Text="{Binding CurrentStatusText}" />
    </StatusBarItem>
</StatusBar>

```

第16章

习题1

为回答这个问题，应看看`Game.cs`文件中的`PlayGame()`方法。查看该方法，列出

- 多少人在玩游戏？他们的名字是什么？
- 当前的玩家是谁？
- 玩家的牌
- 正在处理的牌
- 玩家的操作，例如摸牌、出牌或弃牌
- 被弃的牌列表

- 游戏的状态，例如是否有人获胜

习题2

可将信息存储在数据库中，再检索每个调用所需的数据，使用ASP.NET Session C

关于ASP.NET Session Object的信息，可以阅读这篇文章：

<https://msdn.microsoft.com/en-us/library/ms178581.aspx>

关于VIEWSTATE的信息，可以阅读这篇文章：

<https://msdn.microsoft.com/en-us/library/ms972976.aspx>

第17章

习题1

```
...
using System.Net;
using System.IO;
using Newtonsoft.Json;
using static System.Console;
namespace handofcards
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> cards = new List<string>();
            var playerName = "Benjamin";
            string GetURL =
                "http://handofcards.azurewebsites.net/api/HandOfCards/"
                playerName;
            WebClient client = new WebClient();
            Stream dataStream = client.OpenRead(GetURL);
            StreamReader reader = new StreamReader(dataStream);
            var results =
```



```
        JsonConvert.DeserializeObject<dynamic>(reader.ReadLine())
reader.Close();
foreach (var item in results)
{
    WriteLine((string)item.imageLink);
}
ReadLine();
}
}
}
```

习题2

Web App VM的最大大小是4 CPU/核 (~ 2.6 Ghz) 和7GB的RAM。

标准模式下拥有的VM最大数量是10。高级模式下拥有的VM最大数量是50。这会转化

注意，这用于Web应用程序。可利用Azure VM或Azure云服务获得更多内核和内存。

第18章

习题1

`System.IO`

习题2

需要随机访问文件时，或者不处理字符串数据时，可使用`FileStream`对象写入文件

习题3

- `Peek()`：获取文件中下一个字符的值，但不前移到下一个文件位置

- `Read()`: 获取文件中下一个字符的值，并前移到下一个文件位置
- `Read(char[] buffer, int index, int count)`: 从`buffer[index]`开始
- `ReadLine()`: 获取一行文本
- `ReadToEnd()`: 获取文件中的所有文本

习题4

`DeflateStream`

习题5

- `Changed`: 修改文件时发生

- **Created**: 创建文件时发生
- **Deleted**: 删除文件时发生
- **Renamed**: 重命名文件时发生

习题6

添加一个按钮，切换`FileSystemWatcher.EnableRaisingEvents`属性的值。

第19章

习题1

- (1) 双击**Create Node**按钮，让事件处理程序执行操作。

(2) 在创建XmlComment后，插入如下3行代码：

```
XmlAttribute newPages = document.CreateAttribute("p  
newPages.Value = "1000";  
newBook.Attributes.Append(newPages);
```

习题2

(1) //elements—返回文档中的所有节点。

(2) element—返回文档中的每个元素节点，但不返回元素根节点。

(3) element[@Type='Noble Gas']—返回其Type特性的值是Noble Gas的每

(4) //mass—返回名为mass的所有节点。

(5) `//mass/..`—..使XPath从选中的节点向上移动一个位置，这意味着这个查询

(6) `element/specification[mass='20.1797']`—选择包含mass节点值为20.1797的节点

(7) `element/name[text()='Neon']`—要选择包含测试内容的节点，可使用text()

习题3

XML可以是有效的、格式良好的，也可以是无效的。选择XML文档的一部分时，只是将

习题4

在MainWindow.xaml中添加一个新按钮JSON>XML，然后给MainWindow.xaml.cs

```
private void buttonConvertXMLtoJSON_Click(object sender,
{
```

```

// Load the XML document.
XmlDocument document = new XmlDocument();
document.Load(@"C:\BegVCSharp\Chapter19\XML and Schema
string json = Newtonsoft.Json.JsonConvert.SerializeXml
textBoxResults.Text = json;
System.IO.File.AppendAllText

("C:\BegVCSharp\Chapter19\XML and Schema\Books

}
private void buttonConvertJSONtoXML_Click(object sender,

{

// Load the json document.

string json = System.IO.File.ReadAllText

```

```
(@"C:\BegVCSharp\Chapter19\XML and Schema\Books.json");
```

```
XmlDocument document =
```

```
Newtonsoft.Json.JsonConvert.DeserializeXmlNode(
```

```
textBlockResults.Text =
```

```
FormatText(document.DocumentElement as XmlNode,
```

```
}
```


第20章

习题1

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Sam" };

    var queryResults =
        from n in names
        where n.StartsWith("S")
        orderby n descending

    select n;

    Console.WriteLine("Names beginning with S:");
    foreach (var item in queryResults) {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to exit.");
    Console.ReadLine();
}
```

```
}
```

习题2

在小于5 000 000的集中，没有小于1000的数字：

```
static void Main(string[] args)
{
    int[] arraySizes = { 100, 1000, 10000, 100000,
                        1000000, 5000000, 10000000, 50000000 };
    foreach (int i in arraySizes) {
        int[] numbers = generateLotsOfNumbers(i);
        var queryResults = from n in numbers
                           where n<1000
                           select n;
        Console.WriteLine("number array size = {0}: Count(n<1000)
                           numbers.Length, queryResults.Count()
                           );
    }
}
```

```
    Console.Write("Program finished, press Enter/Return to cont  
    Console.ReadLine();  
}
```

习题3

对于 $n < 1000$ ，性能受到的影响并不明显。

```
static void Main(string[] args)  
{  
    int[] numbers = generateLotsOfNumbers(12345678);  
    var queryResults =  
        from n in numbers  
        where n < 1000  
        orderby n  
  
        select n  
    ;  
    Console.WriteLine("Numbers less than 1000:");  
    foreach (var item in queryResults)
```

```
{
    Console.WriteLine(item);
}
Console.Write("Program finished, press Enter/Return to cont
Console.ReadLine();
}
```

习题4

对于非常大的子集，例如 $n > 1000$ ，而不是 $n < 1000$ ，会非常慢：

```
static void Main(string[] args)
{
    int[] numbers = generateLotsOfNumbers(12345678);
    var queryResults =
        from n in numbers
        where n > 1000

        select n
    ;
```

```

    Console.WriteLine("Numbers less than 1000:");
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }
    Console.Write("Program finished, press Enter/Return to continue");
    Console.ReadLine();
}

```

习题5

会输出所有名字，因为没有查询。

```

static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Sarmiento" };
    var queryResults = names;
    foreach (var item in queryResults) {
        Console.WriteLine(item);
    }
    Console.Write("Program finished, press Enter/Return to continue");
}

```

```
        Console.ReadLine();  
    }
```

习题6

```
static void Main(string[] args)  
{  
    string[] names = { "Alonso", "Zheng", "Smith", "Jor  
"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Sam  
    // only Min() and Max() are available (if no lambda is u  
  
    // for a result set like this consisting only of strings  
    Console.WriteLine("Min(names) = " + names.Min());  
    Console.WriteLine("Max(names) = " + names.Max());  
    var queryResults =  
        from n in names  
        where n.StartsWith("S")  
        select n;  
    Console.WriteLine("Query result: names starting with S")  
    foreach (var item in queryResults)  
    {
```

```
        Console.WriteLine(item);
    }
    Console.WriteLine("Min(queryResults) = " + queryResults.Min());

    Console.WriteLine("Max(queryResults) = " + queryResults.Max());
    Console.WriteLine("Program finished, press Enter/Return to c");
    Console.ReadLine();
}
```

第21章

习题1

注释掉两本书的显式创建代码，替换为提示输入一个新的书名和作者的代码，如下：

```
//Book book = new Book { Title = "Beginning Visual C# 2015",
//                          Author = "Perkins, Reid, and Hammer" };
//db.Books.Add(book);
```

```

//book = new Book { Title = "Beginning XML", Author = "Fawcett
    string title;
    string author;
    Book book;
do
{
    Console.Write("Title: "); title = Console.Read
    Console.Write("Author: "); author = Console.R
    if (!string.IsNullOrEmpty(author))
    {
        book = new Book { Title = title, Author =
        db.Books.Add(book);
        db.SaveChanges();
    }
} while (!string.IsNullOrEmpty(author));

```

习题2

添加一个测试LINQ查询，在添加到数据库之前，看看是否存在书名和作者相同的书。

```

Book book = new Book { Title = "Beginning Visual C# :
    Author = "Perkins, Reid, and Ham

```



```

var testQuery = from b in db.Books
                where b.Title == book.Title && b.Author == bc
                select b;
if (testQuery.Count()<1)
{
    db.Books.Add(book);
    db.SaveChanges();
}

```

习题3

修改生成的类Stock.cs、Store.cs和BookContext.cs，以使用Inventory和I

```

public partial class Stock
{
    ...
    public virtual Store Store { get; set; }
}
public partial class Store
{
    ...
    public Store()

```

```

        {
            Inventory = new HashSet<Stock>();
        }
        ...
        public virtual ICollection<Stock> Inventory

{ get; set; }
    }

    public partial class BookContext : DbContext
    {
        ...
        protected override void OnModelCreating(DbModelBuilder
        {
            modelBuilder.Entity<Book>()
                .HasMany(e => e.Inventory)
                .WithOptional(e => e.Item)
                .HasForeignKey(e => e.Item_Code);
            modelBuilder.Entity<Store>()
                .HasMany(e => e.Inventory)
                .WithOptional(e => e.Store)
                .HasForeignKey(e => e.Store_StoreId);
        }
    }
    class Program
    {

```

```

static void Main(string[] args)
{
    using (var db = new BookContext())
    {
        var query = from store in db.Stores
                    orderby store.Name
                    select store;
        foreach (var s in query)
        {
            XElement storeElement = new XElement("store",
                new XAttribute("name", s.Name),
                new XAttribute("address", s.Address),
                from stock in s.Inventory

                select new XElement("stock",
                    new XAttribute("StockID", stock.StockId),
                    new XAttribute("onHand",
                        stock.OnHand),
                    new XAttribute("onOrder",
                        stock.OnOrder),
                new XElement("book",
                    new XAttribute("title",
                        stock.Item.Title),
                    new XAttribute("author",
                        stock.Item.Author)

```

```

        )// end book
    ) // end stock
); // end store
Console.WriteLine(storeElement);
}

```

习题4

Use the following code:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
namespace BegVCSharp_21_Exercise4_GhostStories
{
    public class Story
    {
        [Key]
        public int StoryID { get; set; }
        public string Title { get; set; }
    }
}

```

```

        public Author Author { get; set; }
        public string Rating { get; set; }
    }
    public class Author
    {
        [Key]
        public int AuthorId { get; set; }
        public string Name { get; set; }
        public string Nationality { get; set; }
    }
    public class StoryContext : DbContext
    {
        public DbSet<Author> Authors { get; set; }
        public DbSet<Story> Stories { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new StoryContext())
            {
                Author author1 = new Author
                {
                    Name = "Henry James",
                    Nationality = "American"
                };
                Story story1 = new Story

```

```

    {
        Title = "The Turn of the Screw",
        Author = author1,
        Rating = "a bit dull"
    };
    db.Stories.Add(story1);
    db.SaveChanges();
    var query = from story in db.Stories
                 orderby story.Title
                 select story;
    Console.WriteLine("Ghost Stories:");
    Console.WriteLine();
    foreach (var story in query)
    {
        Console.WriteLine(story.Title);
        Console.WriteLine();
    }
    Console.WriteLine("Press a key to exit...");
    Console.ReadKey();
}
}

```

第22章

习题1

上述应用程序都可以。

习题2

实现数据协定，需要DataContractAttribute和DataMemberAttribute特性。

习题3

使用.svc扩展。

习题4

这是一种方式，但把所有WCF配置放在一个独立配置文件中通常很简单，例如web.c

习题5

```
[ServiceContract]
public interface IMusicPlayer
{
    [OperationContract(IsOneWay=true)]
    void Play();
    [OperationContract(IsOneWay=true)]
    void Stop();
    [OperationContract]
    TrackInformation GetCurrentTrackInformation();
}
```

还需要一个数据协定来封装跟踪信息，在上面的代码中就是TrackInformation。

第23章

习题1

(1) 修改BlankPage1页面的XAML，如下：

```
<Page
    x:Class="BasicNavigation.BlankPage1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BasicNavigation"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Loaded="Page_Loaded">
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <CommandBar>
            <AppBarToggleButton x:Name="toggleButtonBold" Icon="Bold"
Click="AppBarToggleButtonBold_Click" />
            <AppBarSeparator />
            <AppBarButton Icon="Back" Label="Back" Click="buttonGoBack_Click" />
            <AppBarButton Icon="Forward" Label="Forward" Click="AppBarButtonForward_Click" />
        </CommandBar>
    </Grid>
</Page>
```

```

        <CommandBar.SecondaryCommands>
            <AppBarButton Icon="Camera" Label="Take picture" />
            <AppBarButton Icon="Help" Label="Help" />
        </CommandBar.SecondaryCommands>
    </CommandBar>

    <TextBlock x:Name="textBlockCaption" Text="Page 1" Horizontal
Margin="10,50,10,10" VerticalAlignment="Top"/>

    <StackPanel Orientation="Horizontal" Grid.Row="1" HorizontalA
VerticalAlignment="Bottom">
        <Button Content="Page 2" Click="buttonGoto2_Click" />
        <Button Content="Page 3" Click="buttonGoto3_Click" />
        <Button Content="Back" Click="buttonGoBack_Click" />
    </StackPanel>

    <WebView x:Name="webViewControl" HorizontalAlignment="Stret
VerticalAlignment="Stretch" />
</Grid>
</Page>

```

(2) 进入代码隐藏代码，添加如下代码行：

```

webViewControl.Navigate(new Uri("http://www.wrox.com"));

Application.Current.Resuming += (sender, o) => webViewControl.
Uri("http://www.amazon.com/Beginning-Visual-C-2015-Programming/dp
=UTF8&qid=1444947234&sr=8-1&keywords=beginning+visual+c%23+2015")

```

习题2

在Capabilities选项卡上指定应用程序有哪些功能（在Package.appxmanifest

习题3

- （1）创建一个新的通用应用程序项目。
- （2）把一个Pivot控件拖放到设计视图上。
- （3）修改第一个PivotItem，如下：

```
<PivotItem Header="Wrox Homepage">  
    <Grid>  
        <WebView Name="WebViewControl" />
```

```
        </Grid>
    </PivotItem>
```

(4) 修改第二个PivotItem, 如下:

```
<PivotItem Header="Hello Pivot!">
    <Grid>
        <TextBlock Text="Hello Pivot!" HorizontalAlignment="Center"
VerticalAlignment="Center" />
    </Grid>
</PivotItem>
```

(5) 添加第三个PivotItem, 如下:

```
<PivotItem Header="Wrox Logo">
    <Grid>
        <Image Source="http://media.wiley.com/assets/253/59/wro
    </Grid>
</PivotItem>
```

(6) 最后, 在页面的构造函数中调用Navigate, 把Web查看控件导航到所选的页面

```
webViewControl.Navigate(new Uri("http://www.wrox.com"));
```